Understanding Assembly Language

Dennis Yurichev

Understanding Assembly Language

(Reverse Engineering für Einsteiger)

Why two titles? Read here: on page xi.

Dennis Yurichev
my emails



Text-Version (23. Oktober 2023).

Die aktuellste Version (und eine russische Ausgabe) dieses Textes ist auf https://beginners.re/ verfügbar. Eine Version für den E-Book-Leser ist dort ebenfalls erhältlich.

Übersetzer gesucht!

Vielleicht möchten Sie mir bei der Übersetzung dieser Arbeit in andere Sprachen (außer Englisch und Russisch) helfen. Senden Sie mir einen übersetzten Textteil, egal wie kurz und ich arbeite ihn in den LaTeX-Quellcode ein.

Hier lesen.

Geschwindigkeit ist nicht wichtig, denn im Endeffekt ist es ein Open-Source-Projekt. Ihr Name wird als Mitwirkender des Projekts erwähnt. Koreanisch, Chinesisch und Persisch sind für Verleger reserviert. Eine englische und russische Version mache ich selber, allerdings ist mein Englisch immer noch schrecklich. Ich bin also dankbar über alle Anmerkungen bezüglich der Grammatik und Ähnlichem. Auch mein Russisch ist teilweise fehlerhaft, also bin ich auch hier dankbar über Anmerkungen!

Also zögern Sie nicht mir zu schreiben: my emails.

Inhaltsverzeichnis (gekürzt)

Inhaltsverzeichnis

[1]Floating-Point Unit

[2]Java Virtual Machine
[3]Pseudozufallszahlen-Generator

Vorwort

Warum zwei Titel?

Dieses Buch hieß von 2014-2018 "Reverse Engineering for Beginners", jedoch hatte ich immer die Befürchtung, dass es den Leserkreis zu sehr einengen würde.

Infosec Leute kennen sich mit "Reverse Engineering" aus, jedoch hörte ich selten das Wort "Assembler" von Ihnen.

Desweiteren ist der Begriff "Reverse Engineering" etwas zu kryptisch für den Großteil von Programmierern, diesen ist jedoch "Assembler" geläufig.

Im Juli 2018 änderte ich als Experiment den Titel zu "Assembly Language for Beginners" und veröffentlichte den Link auf der Hacker News-Website[4]. Das Buch kam allgemein gut an.

Aus diesem Grund hat das Buch nun zwei Titel.

Ich habe den zweiten Titel zu "Understanding Assembly Language" geändert, da es bereits eine Erscheinung mit dem Titel "Assembly Language for Beginners" gab. Einige Leute sind der Meinung, dass "for Beginners" etwas sarkastisch klingt, für ein Buch mit ~1000 Seiten.

Die beiden Bücher unterscheiden sich lediglich im Titel, dem Dateinamen (UAL-XX.pdf beziehungsweise RE4B-XX.pdf), URL und ein paar der einleitenden Seiten.

Über Reverse Engineering

Es gibt verschiedene verbreitete Interpretationen des Begriffs Reverse Engineering:
1) Reverse Engineering von Software: Rückgewinnung des Quellcodes bereits kompilierter Programme;
2) Das Erfassen von 3D Strukturen und die digitalen Manipulationen die zur Duplizierung notwendig sind;
3) Nachbilden von DBMS[5]-Strukturen.
Dieses Buch behandelt die erste Interpretation.

Voraussetzungen

Grundlegende Kenntnisse der Programmiersprache C. Empfohlene Literatur: 11.1.3 on page 677.

Übungen und Aufgaben

…befinden sich nun alle auf der Website: `http://challenges.re`.

Lob für

`https://beginners.re/#praise`.

[4] `https://news.ycombinator.com/item?id=17549050`
[5] Database Management Systems

Danksagung

Für das geduldige Beantworten aller meiner Fragen: SkullC0DEr.

Für Anmerkungen über Fehler und Unstimmigkeiten: Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin "netch" Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin[6], Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Pavel Shakhov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon[7], Ben L., Etienne Khan, Norbert Szetei[8], Marc Remy, Michael Hansen, Derk Barten, The Renaissance[9], Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista, Ashod Nakashian..

Für die Hilfe in anderen Dingen: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha "tiphareth" Verbitsky.

Für die Übersetzung des Buchs ins Vereinfachte Chinesisch: Antiy Labs (antiy.cn), Archer.

Für die Übersetzung des Buchs ins Koreanische: Byungho Min.

Für die Übersetzung des Buchs ins Niederländische: Cedric Sambre (AKA Midas).

Für die Übersetzung des Buchs ins Spanische: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevarena.

Für die Übersetzung des Buchs ins Portugiesische: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini.

Für die Übersetzung des Buchs ins Italienische: Federico Ramondino[10], Paolo Stivanin[11], twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro[12], bluepulsar.

Für die Übersetzung des Buchs ins Französische: Florent Besnard[13], Marc Remy[14], Baudouin Landais, Téo Dacquet[15], BlueSkeye@GitHub[16].

Für die Übersetzung des Buchs ins Deutsche: Dennis Siekmeier[17], Julius Angres[18], Dirk Loser[19], Clemens Tamme, Philipp Schweinzer, Tobias Deiminger.

[6] goto-vlad@github
[7] https://github.com/pixjuan
[8] https://github.com/73696e65
[9] https://github.com/TheRenaissance
[10] https://github.com/pinkrab
[11] https://github.com/paolostivanin
[12] https://github.com/Internaut401
[13] https://github.com/besnardf
[14] https://github.com/mremy
[15] https://github.com/T30rix
[16] https://github.com/BlueSkeye
[17] https://github.com/DSiekmeier
[18] https://github.com/JAngres
[19] https://github.com/PolymathMonkey

Für die Übersetzung des Buchs ins Polnische: Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski.

Für die Übersetzung des Buchs ins Japanische: shmz@github[20],4ryuJP@github[21].

Für das Korrekturlesen: Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev[22] der unglaublich viel Arbeit in die Korrektur vieler Fehler investiert hat.

Danke auch an alle, die auf github.com Anmerkungen und Korrekturen eingebracht haben.

Es wurden viele LaTeXPakete genutzt: Vielen Dank an deren Autoren.

Donors

Dank an diejenigen die mich während der Zeit in der ich wichtige Teile des Buchs geschrieben habe unterstützt haben:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the R0ck ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov ($3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous ($15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300), Ernesto Bonev Reynoso ($30).

Vielen Dank an alle Spender!

Mini-FAQ

F: Was sind die Voraussetzungen die der Leser dieses Buchs erfüllen sollte?

A: Grundlagenwissen der Programmiersprachen C und / oder C++ sind wünschenswert.

F: Sollte ich wirklich x86/x64/ARM und MIPS auf einmal lernen? Ist das nicht zuviel?

[20] https://github.com/shmz
[21] https://github.com/4ryuJP
[22] https://vasil.ludost.net/

A: Anfänger können erstmal nur über x86/x64 lesen und den ARM- und MIPS-Teil überspringen oder überfliegen.

F: Kann ich eine russische oder englische Version als Druckausgabe kaufen?

A: Leider nicht, bisher ist kein Verleger an einer russischen oder englischen Version interessiert. Bis es soweit ist, können Sie Ihren Lieblings-Copy-Shop bitten es zu drucken und zu binden. https://yurichev.com/news/20200222_printed_RE4B/.

F: Gibt es eine EPUB- oder MOBI-Version?

A: Dieses Buch ist in hohem Maße abhängig von TeX- / LaTeX-spezifischen Techniken, was das Konvertieren zu HTML schwierig macht (EPUB und MOBI basieren auf HTML).

F: Warum sollte ich heutzutage noch Assembler lernen?

A: Falls Sie kein BS[23]-Entwickler sind, werden Sie vermutlich nie in Assembler programmieren müssen — aktuelle Compiler (2010 und später) können sehr viel besser optimieren als Menschen[24].

Auch sind aktuelle CPU[25]s sehr komplexe Komponenten und Wissen über Assembler hilft nicht wirklich um die Interna zu verstehen.

Davon abgesehen, gibt es mindestens zwei Bereiche in denen ein gutes Verständnis von Assembler hilfreich sein kann: Zuallererst, bei der Security- und Malware-Forschung, aber auch um ein besseres Verständnis des kompilierten Codes zu bekommen. Dieses Buch ist somit für diejenigen geschrieben, die Assembler eher verstehen als darin programmieren wollen. Das ist der Grund, warum viele Ausgabe-Beispiele des Compilers in diesem Buch enthalten sind.

F: Ich habe in der PDF-Datei auf einen Link geklickt. Wie komme ich zurück?

A: Im Adobe Acrobat Reader geht dies durch betätigen von Alt+CursorLinks. In Evince durch die "<"-Taste.

F: Darf ich dieses Buch drucken / für Lehrzwecke benutzen?

A: Selbstverständlich! Das ist der Grund warum es unter der Creative Commons Lizenz (CC BY-SA 4.0) veröffentlicht wird.

F: Warum ist dieses Buch kostenlos? Du hast gute Arbeit geleistet. Das ist verdächtig, wie bei vielen anderen kostenlosen Dingen.

A: Meiner Erfahrung nach schreiben Autoren von technischer Literatur diese des Lernens willen. Es ist nicht möglich angemessen viel Geld hierfür zu bekommen.

F: Wie kann man einen Job im Bereich des Reverse Engineering bekommen?

A: Von Zeit zu Zeit gibt es Threads zu Jobangeboten auf [26]. Versuchen Sie es dort einmal.

Ein ähnlicher Job-Thread ist unter„netsec" subreddit zu finden.

---

[23] Betriebssystem

[24] Ein lesenswerter Artikel zu diesem Thema: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

[25] Central Processing Unit

[26] reddit.com/r/ReverseEngineering/

F: Ich habe eine Frage...

A: Senden Sie sie mir per E-Mail (my emails).

Über die koreanische Übersetzung

Im Januar 2015 hat die Acorn Publishing Company (www.acornpub.co.kr) in Süd-Korea viel Aufwand in die Übersetzung und Veröffentlichung meines Buchs ins Koreanische (mit Stand August 2014) investiert.

Es ist jetzt unter dieser Webseite verfügbar.

Der Übersetzer ist Byungho Min (twitter/tais9). Die Cover-Gestaltung wurde von meinem künstlerisch begabten Freund Andy Nechaevsky erstellt: facebook/andydinka. Die Acorn Publishing Company besetzt die Urheberrechte an der koreanischen Übersetzung.

Wenn Sie also ein *echtes* Buch in Ihrem Buchregal auf koreanisch haben und mich bei meiner Arbeit unterstützen wollen, können Sie das Buch nun kaufen.

Über die persische Übersetzung (Farsi)

In 2016 wurde das Buch von Mohsen Mostafa Jokar übersetzt, der in der iranischen Community auch für die Übersetzung des Radare Handbuchs[27] bekannt ist). Es ist auf der Homepage des Verlegers[28] (Pendare Pars) verfügbar.

Hier ist ein Link zu einem 40-seitigen Auszug: `https://beginners.re/farsi.pdf`.

National Library of Iran registration information: `http://opac.nlai.ir/opac-prod/bibliographic/4473995`.

Über die chinesische Übersetzung

Im April 2017 wurde die chinesische Übersetzung von Chinese PTPress fertiggestellt, bei denen auch das Copyright liegt.

Die chinesische Version kann hier bestellt werden: `http://www.epubit.com.cn/book/details/4174`. Ein Auszug und die Geschichte der Übersetzung kann hier gefunden werden: `http://www.cptoday.cn/news/detail/3155`.

Der Hauptübersetzer ist Archer, dem der Autor sehr iel verdankt. Archer war extrem akribisch (im positiven Sinne) und meldete den Großteil der bekannten Fehler, was für Literatur wie dieses Buch extrem wichtig ist. Der Autor empfiehlt diesen Service jedem anderen Autor!

Die Mitarbeiter von Antiy Labs halfen ebenfalls bei der Übersetzung. Hier ist das Vorwort von ihnen.

[27] `http://rada.re/get/radare2book-persian.pdf`
[28] `http://goo.gl/2Tzx0H`

Kapitel 1

Code-Muster

1.1 Die Methode

Als der Autor dieses Buches zunächst C und später C++ lernte, schrieb er kleine Codestücke, kompilierte sie und untersuchte anschließend den Assembler-Code des Compilers. Dies machte es sehr einfach zu verstehen, was in dem Code den er geschrieben hatte passierte. [1] Er hat dies so oft getan, dass der Zusammenhang zwischen dem C/C++-Code und dem was der Compiler daraus macht tief in seinem Verstand verankert ist. So ist es einfach sich schnell einen Überblick über das Aussehen und die Funktion einer C-Quelle zu verschaffen. Vielleicht ist diese Vorgehensweise auch für andere hilfreich.

Übrigens gibt es eine hervorragende Webseite auf der dasselbe mit verschiedenen Compilern getan werden kann, anstatt diese zu installieren. Auch diese kann genutzt werden: `https://godbolt.org/`.

Übungen

Als der Autor dieses Buches Assembler erlernte, hat er oft kleine C-Funktionen kompiliert und anschließend nach und nach in Assembler nachprogrammiert um den Code so klein wie möglich zu machen. Dies ist in einer Anwendung in der Praxis heutzutage vielleicht nicht mehr sinnvoll, weil moderne Compiler in der Regel weitaus besser optimieren können als ein Mensch. Dennoch ist es ein guter Weg um Assembler besser zu verstehen. Versuchen Sie ruhig einmal einen Assembler-Quelltext aus dem Buch bei gleicher Funktionalität kürzer zu schreiben. Vergessen Sie aber nicht Dinge die Sie programmiert haben zu testen.

[1] Tatsächlich tut er dies immer noch wenn er einen bestimmten Code-Teil nicht versteht.

Optimierungsstufen und Debug-Informationen

Quellcode kann von verschiedenen Compilern mit verschiedenen Optimierungsstufen übersetzt werden. Üblicherweise hat ein Compiler drei solcher Stufen, wobei Stufe Null eine deaktivierte Optimierung bedeutet. Die Optimierung kann sich ebenso auf die Code-Größe als auch auf die Ausführgeschwindigkeit beziehen. Ein Nicht-optimierender Compiler ist schneller und erstellt einfacher zu verstehenden (aber auch längeren) Code. Demgegenüber ist ein optimierender Compiler langsamer und versucht Code zu erstellen, dessen Ausführgeschwindigkeit höher, aber nicht notwendigerweise kompakter, ist. Zusätzlich zu diesen Optimierungsmöglichkeiten kann ein Compiler Informationen in den Code einfügen, die das spätere Debuggen vereinfachen. Eine wichtige Eigenschaft des Debug-Codes ist, dass er gegebenenfalls Links zwischen den Zeilen des Quellcodes und den entsprechenden Maschinen-Code-Adressen enthält. Optimierende Compiler hingegen tendieren dazu Code zu erzeugen, der ganze Zeilen des Quellcodes wegoptimiert, die dann dementsprechend im Maschinencode nicht auftauchen. Ein Reverse Engineer kann beiden Varianten begegnen, einfach, weil einige Software-Entwickler die Optimierung des Compilers nutzen und andere nicht. Aufgrund dieser Tatsache werden Sie in diesem Buch auch Beispielcode für optimierten und nichtoptimierten Compiler-Code finden.

1.2 Einige Grundlagen

1.2.1 Eine kurze Einführung in die CPU

Die CPU ist die Komponente, die den Maschinencode ausführt aus dem ein Programm besteht.

Ein kurzes Glossar:

Befehl : Ein einfaches CPU-Kommando. Die einfachsten Beispiele hierfür sind: Verschieben von Daten zwischen Registern, Arbeiten mit Speicher, einfache arithmetische Operationen. In der Regel hat jede CPU ihre eigene Befehlssatz-Architektur (ISA[2]).

Maschinencode : Code den die CPU direkt verarbeitet. Jeder Befehl ist in der Regel durch ein paar Byte kodiert.

Assembler-Sprache : Mnemonics und einige Makro-ähnliche Erweiterungen um das Leben der Programmierer zu erleichtern.

CPU-Register : Jede CPU hat eine feste Anzahl von Mehrzweck-Registern (GPR[3]). ≈ 8 bei x86, ≈ 16 bei x86-64, ≈ 16 bei ARM. Die einfachste Möglichkeit um Register zu verstehen, ist sie als typlose, temporäre Variable zu betrachten. Stellen Sie sich vor Sie würden mit einer Hochsprache arbeiten und könnten nur acht 32-Bit (oder 64-Bit) Variablen nutzen. Dennoch ist damit eine Menge möglich!

Man kann sich jetzt fragen, warum die Unterscheidung zwischen Maschinencode und einer Hochsprache notwendig ist. Die Antwort liegt einfach in der Tatsache, dass

[2] Instruction Set Architecture
[3] General Purpose Registers

Menschen und CPUs nicht gleich sind—Es ist sehr viel einfacher für Menschen eine Programmiersprache wie C/C++, Java, Python, usw. zu lesen. Für eine CPU jedoch ist es einfacher eine geringere Abstraktion zu verarbeiten.

Vielleicht wäre es möglich eine CPU zu entwickeln die direkt eine Hochsprache ausführen kann, diese wäre aber sehr viel komplexer als es heute der Fall ist. In ähnlicher Weise ist es für Menschen äußerst unkomfortabel in einer Assembler-Sprache zu programmieren. Diese ist sehr hardwarenah und kaum zu realisieren ohne schwer zu findende Fehler zu machen. Das Programm, welches die Hochsprache in Assembler konvertiert nennt sich *Compiler* oder Übersetzer.

Einige Anmerkungen zu den verschiedenen ISAs

Die x86 ISA hatte immer Opcodes variabler Länge. Als die 64-Bit-Ära aufkam beeinflussten deren Erweiterungen die Befehlssatz-Architektur nicht sehr stark. Tatsächlich enthält die x86-Architektur immer noch viele Befehle, die zunächst in den 16-bit 8086 CPU implementiert wurden und noch immer in aktuellen CPU enthalten sind. ARM ist eine RISC[4] CPU mit einer konstanten Opcode-Länge, die in der Vergangenheit einige Vorteile aufwies. In den Anfängen waren alle ARM-Befehle in vier Byte kodiert[5]. Dies ist nun als „ARM-Mode“ bekannt. Irgendwann dachte ARM, dass dieser Vorgehensweise nicht so sparsam war wie zuerst angenommen. Tatsächlich, benötigen die meisten Befehle in der Praxis[6] weniger Platz für die Kodierung. Darum wurde ein Befehlssatz, genannt Thumb, eingeführt, in dem jeder Befehl in zwei Byte kodiert wird. Dies ist nun als „Thumb-Mode“ bekannt. Trotzdem können nicht *alle* ARM-Befehle in nur zwei Byte kodiert werden, was den Thumb Befehlssatz in gewisser Weise einschränkt. Erwähnenswert ist, dass Code der für ARM- und Thumb-Mode kompiliert wurde in einem einzelnen Programm vermischt sein kann. Die ARM-Erfinder beschlossen den Thumb-Mode zu erweitern, was zu Thumb-2 führte, der in ARMv7 auftaucht. Thumb-2 nutzt immer noch 2-Byte-Befehle, enthält aber zusätzlich auch einige 4-Byte-Befehle. Ein verbreitetes Missverständnis ist, dass Thumb-2 eine Mischung aus und Thumb und ARM ist. Dies ist falsch. Stattdessen wurde Thumb-2 erweitert um alle Prozessor-Eigenschaften zu unterstützen, so dass er mit dem ARM-Mode konkurrieren kann—ein Ziel, dass klar erreicht wurde, da der Großteil der iPod/iPhone/iPad mit dem Thumb-2-Befehlssatz kompiliert werden (zugegebenermaßen, auch durch die Tatsache, dass Xcode dies als Standard-Einstellung so macht). Später erschienen die 64-bit ARM. Diese ISA hat 4-Byte-Opcodes und benötigt den traditionellen Thumb-Mode nicht mehr. Die 64-Bit-Anforderungen beeinflussten die ISA, was uns nun zu drei ARM-Befehlssätzen führt: ARM-Mode, Thumb-Mode (inklusive Thumb-2) und ARM64. Diese Befehlssätze überschneiden sich teilweise, dennoch sind sie eher eigenständig als Variationen eines einzelnen Befehlssatz. Aus diesem Grund wird versucht in diesem Buch von allen drei Varianten Code-Beispiele zu zeigen. Es gibt übrigens noch eine Menge anderer RISC ISAs mit einer konstanten Opcode-Länge von 32 Bit, wie MIPS, PowerPC und Alpha AXP.

[4] Reduced Instruction Set Computing

[5] Befehle mit fester Länge sind ürigens so praktisch weil die nächste (oder vorherige) Befehlsadresse ohne großen Aufwand berechnet werden kann. Diese Eigenschaft wird im Kapitel switch() diskutiert.

[6] zum Beispiel MOV/PUSH/CALL/Jcc

1.2.2 Zahlensysteme

> Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.
>
> Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Menschen sind an das Dezimal-System gewöhnt, möglicherweise weil sie zehn Finger haben. Trotzdem hat die Zahl 10 keine besondere Bedeutung in der Wissenschaft und Mathematik. Das natürliche Zahlensystem in der Digitaltechnik ist binär: 0 für die Abwesenheit und 1 für die Anwesenheit von Strom in einer Leitung. Die binäre 10 ist 2 im Dezimalsystem; Die binäre 100 ist 4 im Dezimalsystem und so weiter.

Wenn ein Zahlensystem 10 Ziffern hat, spricht man von *Basis* von 10. Binäre Zahlensysteme haben die *Basis* von 2.

Wichtige Dinge zum Merken:

1. *Nummer* ist eine Nummer, während *Ziffer* in der Regel eine einzelne Zahl ist;
2. Eine Zahl ändert sich nicht beim Konvertieren in ein anderes Zahlensystem: nur die Darstellung ist anders.

Wie konvertiert man eine Zahl von einer Basis in eine andere?

Fast überall wird das Stellenwertsystem genutzt. Dies bedeutet, dass eine einzelne Ziffer je nach Position in der Zahl ein bestimmtes Gewicht hat. Wenn 2 an der rechten Position steht, ist es eine 2. Wenn sie jedoch eine Position weiter links steht, ist die Zahl eine 20.

Wofür steht 1234?

$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4$ = 1234 oder $1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$

Das Gleiche gilt für binäre Zahlen, nur zur Basis 2 statt 10. Wofür steht 0b101011?

$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43$ oder $32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$

Stellenwertsysteme können den Additionssystemen wie den römischen Ziffern gegenübergestellt werden [7]. Möglicherweise hat die Menschheit zu Stellenwertsystemen gewechselt, weil diese für einfache Operationen (Addition, Multiplikation, usw.) per Hand oder auf Papier einfacher zu verwenden sind.

Tatsächlich können binäre Zahlen genauso addiert, subtrahiert und so weiter werden wie man es in der Schule gelernt hat, es stehen allerdings nur zwei Ziffern zur Verfügung.

Binärzahlen sind sperrig, wenn sie im Quellcode oder Speicherauszügen auftreten. Hier bietet sich die Verwendung des Hexadezimalsystems an. Die Basis besteht hier aus den Ziffen 0..9 und den sechs lateinischen Buchstaben A..F. Jede hexadezimale

[7] Zur Entwickling der Zahlensysteme, siehe [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

Ziffer besteht aus vier Bits oder vier binären Ziffern, was die Konvertierung zwischen Hexadezimal und Binär, selbst im Kopf, sehr einfach macht.

Hexadezimal	Binär	Dezimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Woran sieht man welche Basis gerade verwendet wird?

Dezimalzahlen werden für gewöhnlich ohne Zusatz geschrieben, zum Beispiel 1234. Einige Assembler betonen die Basis 10 jedoch mit dem Zusatz ”d”: 1234d.

Binärzahlen sind manchmal mit dem Präfix ”0b” gekennzeichnet: 0b100110111 (GCC[8] hat eine Nicht-standardisierte Erweiterung hierfür[9]). Ein weiterer Weg ist der ”b” Suffix, zum Beispiel: 100110111b. In diesem Buch wird versucht durchgängig die ”0b”-Präfix-Variante zu benutzen.

Hexadezimalzahlen werden in C/C++ und anderen Hochsprachen mit dem ”0x”-Präfix versehen: 0x1234ABCD, oder sie haben einen ”h”-Suffix: 1234ABCDh - dies ist eine weit verbreitete Variante in Assembler und Debuggern. Wenn die Zahl mit A..F startet, muss eine 0 davor geschrieben werden: 0ABCDEFh. In diesem Buch wird versucht durchgängig die ”0x”-Präfix-Variante zu benutzen.

Ist es ratsam das Konvertieren von Zahlen im Kopf zu Üben? Die Tabelle der einstelligen Hexadezimalziffern kann leicht auswendig gelernt werden. Für größere Zahlen lohnt sich der Aufwand vielleicht nicht wirklich.

Oktalsystem

Dies ist ein weiteres Zahlensystem, welches in der Computerprogrammierung sehr verbreitet ist: die 8 Ziffern (0..7) sind jeweils drei Bits zugeordnet. Dies macht das Konvertieren in andere Zahlensysteme sehr einfach. Das Oktalsystem wurde fast

[8] GNU Compiler Collection
[9] https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html

überall ersetzt, dennoch gibt es, überraschenderweise, ein *NIX-Tool, welches häufig genutzt wird und eine Oktalzahl als Aufrufparameter hat: `chmod`.

Wie viele *NIX-Nutzer wissen, ist der Aufrufparameter von `chmod` eine Zahl mit drei Ziffern. Die erste beschreibt die Rechte des Besitzers einer Datei, die zweite der Gruppe zu der die Datei gehört und die dritte ist für alle anderen. Jede Ziffer kann in ihrer binären Form repräsentiert werden:

Dezimal	Binär	Bedeutung
7	111	**rwx**
6	110	**rw-**
5	101	**r-x**
4	100	**r--**
3	011	**-wx**
2	010	**-w-**
1	001	**--x**
0	000	**---**

Jedes Bit wird also abgebildet auf die Flags: lesen/schreiben/ausführen.

Der Grund warum hier das `chmod`-Kommando erwähnt wird, ist weil der Aufrufparameter auch in Oktalform angegeben werden kann. Zum Beispiel die 644: wenn `chmod 644 file` ausgeführt wird, werden Lese/Schreib-Rechte für den Besitzer, Lese-Rechte für die Gruppe und ebenfalls Lese-Rechte für alle anderen gesetzt.

Die Oktalzahl 644 ist in binärer Form `110100100`, oder (in Dreierbitgruppen) `110 100 100`.

Man erkennt nun sehr schön, dass jedes dieser Dreiergruppen die Rechte für Besitzer/Gruppe/Andere beschreibt: zuerst `rw-`, dann `r--` und zuletzt nochmals `r--`.

Das Oktalsystem war sehr populär auf alten Computerarchitekturen wie PDP-8, weil ein Word hier 12, 24 oder 36 Bit breit sein konnte, und diese Zahlen alle durch drei teilbar sind. Daher war die Verwendung des Oktalsystems am Natürlichsten. Heutzutage sind die Word- und Adressbreiten der Computer 16, 32 oder 64 Bit, und damit durch vier teilbar. Dementsprechend ist die Verwendung des Hexadezimalsystems natürlicher und intuitiver.

Das Oktalsystem wird von allen standardkonformen C/C++-Compilern unterstützt. Dies kann gelegentlich zu Verwirrung führen, weil Oktalzahlen mit einer vorangestellten Null gekennzeichnet werden, also zum Beispiel 0377, was 255 in der Dezimalschreibweise entspricht. Ein versehentlicher Schreibfehler wie "09" statt "9" wird der Compiler erkennen, weil die 9 keine Ziffer des Oktalsystems ist. GCC meldet einen Fehler wie:
`error: invalid digit "9" in octal constant`.

Teilbarkeit

Wenn Sie eine Dezimalzahl wie 120 sehen, ist schnell ersichtlich, dass diese durch 10 teilbar ist, weil die letzte Ziffer eine Null ist. Genauso ist 123400 durch 100 teilbar,

weil die letzten zwei Ziffern Nullen sind.

Auf die gleiche Weise ist die Hexadezimalzahl 0x1230 durch 0x10 (oder 16) und 0x123000 durch 0x1000 (oder 4096) teilbar.

Die Binärzahl 0b1000101000 ist durch 0b1000 (8) teilbar und so weiter. Diese Eigenschaft kann häufig dazu genutzt werden um schnell herauszufinden ob ein Speicherinhalt zu einer bestimmten Grenze aufgefüllt ist. Beispielsweise starten Sektionen in PE[10]-Dateien fast immer an Adressen die in hexadezimaler Schreibweise mit drei Nullen enden: 0x41000, 0x10001000, usw. Der Grund dafür ist, dass fast alle diese Sektionen an Grenzen ausgerichtet sind, die Vielfache von 0x1000 (4096) Byte sind.

Langzahlarithmetik und Basis

Langzahlarithmetik nutzt sehr große Zahlen die in mehreren Bytes gespeichert sein können. Der öffentliche und private Schlüssel im RSA-Algorithmus beispielsweise benötigt 4096 Bit und mehr. German text placeholder

Aussprache

Nummer mit nicht-dezimaler Basis werden in der Regel ziffernweise gelesen “eins-null-null-eins-eins-...”. Wörter wie “zehn”, “tausend”, und so weiter werden meist nicht so genannte, um Verwechslungen mit dem Dezimalsystem zu vermeiden.

Fließkommazahlen

Um Fließkommazahlen von Integer unterscheiden zu können, wird ihnen oft eine “.0” angehängt, zum Beispiel 0.0, 123.0, und so weiter.

1.3 Leere Funktion

Die denkbar einfachste Funktion ist sicher eine die nichts macht:

Listing 1.1: C/C++-Code

```
void f()
{
        return;
};
```

Kompilieren wir diesen Quellcode!

1.3.1 x86

Nachfolgende die Ausgabe die sowohl der optimierende GCC als auch der MSVC-Compiler auf einer x86-Plattform produziert:

[10] Portable Executable

Listing 1.2: Optimierender GCC/MSVC (Assemblercode)

```
f:
        ret
```

Es gibt lediglich eine Anweisung RET, welche die Ausführung zurück an caller übergibt.

1.3.2 ARM

Listing 1.3: Optimierender Keil 6/2013 (ARM Modus) Assemblercode

```
f       PROC
        BX      lr
        ENDP
```

Die Rücksprungadresse ist im ARM-ISA nicht auf dem lokalen Stack gesichert sondern im Link-Register. Die BX LR-Anweisung führt dazu, dass die an diese Adresse gesprungen wird—-also die Ausführung wieder an den caller übergeben wird.

1.3.3 MIPS

Es gibt zwei Namenskonventionen für Register in der MIPS-Welt: durch Zahlen (von$0 bis $31) oder Pseudonamen ($V0, $A0, usw.).

Die Assembler-Ausgabe von GCC unten listet die Register mit Nummern auf:

Listing 1.4: Optimierender GCC 4.4.5 (Assemblercode)

```
        j       $31
        nop
```

…während IDA[11] Pseudonamen nutzt:

Listing 1.5: Optimierender GCC 4.4.5 (IDA)

```
        j       $ra
        nop
```

Die erste Anweisung ist die Sprunganweisung die die Ausführung durch Springen an die Adresse in Register $31 (oder $RA) wieder an den caller übergibt.

Dies ist das Register analog zu LR[12] in ARM.

Die zweite Anweisung ist NOP[13] und macht gar nichts; sie kann hier erst mal ignoriert werden.

[11] Interaktiver Disassembler und Debugger entwickelt von Hex-Rays
[12] Link Register
[13] No Operation

Eine Anmerkung zu MIPS-Anweisungen und Registernamen

Register und Anweisungsnamen sind in Bezug auf MIPS traditionellerweise klein geschrieben. Um einen einheitlichen Stil zu haben, werden in diesem Buch jedoch Großbuchstaben genutzt. Dies gilt für alle ISAs die in diesem Buch besprochen werden.

1.3.4 Leere Funktionen in der Praxis

Abgesehen von der Tatsache, dass leere Funktionen nutzlos sind, begegnet man ihnen in Low-Level-Code häufiger.

Zum Einen sind Debugging-Funktionen wie die Folgende recht verbreitet:

Listing 1.6: C/C++-Code

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
        // open log file
        // write to log file
        // close log file
#endif
};

void some_function()
{
        ...

        dbg_print ("we did something\n");

        ...
};
```

Bei einem Nicht-Debug-Build (d.h. “Release”), ist `_DEBUG` nicht definiert. Die Funktion `dbg_print()` ist, obwohl sie bei der Ausführung aufgerufen wird, ohne Inhalt.

Eine verbreitete Art des Software-Schutzes ist es verschiedene Build-Versionen zu erstellen: eine für legale Kunden und eine Demo-Version. Der Letzteren können zum Beispiel wichtige Funktionen fehlen, wie folgt:

Listing 1.7: C/C++-Code

```
void save_file ()
{
#ifndef DEMO
        // a real saving code
#endif
};
```

Die Funktion`save_file()` kann vom Nutzer über den Menüpunkt `File->Save` aufgerufen werden. Die Demo-Version könnte mit einem deaktivierten Menüpunkt ausgeliefert werden, selbst wenn ein Cracker in der Lage ist diesen wieder zu aktivieren, wird die leere Funktion ohne sinnvollen Inhalt ausgeführt

IDA markiert solche Funktionen mit Namen wie `nullsub_00`, `nullsub_01`, usw.

1.4 Die einfachste Funktion

Die einfachste, mögliche Funktion ist vermutlich eine, die lediglich einen konstanten Wert zurückgibt.

Hier ist sie:

Listing 1.8: C/C++

```
int f()
{
        return 123;
};
```

Und nun in kompilierter Version!

1.4.1 x86

Nachfolgend das, was sowohl der optimierende GCC- als auch MSVC-Compiler auf einer x86-Plattform erzeugt:

Listing 1.9: Optimierender GCC/MSVC (Assemblercode)

```
f:
        mov     eax, 123
        ret
```

Es gibt zwei Anweisungen: die erste platziert den Wert 123 in das EAX-Register, welches per Konvention als Speicherplatz für den Rückabewert genutzt wird. Die zweite ist RET, die die Ausführung wieder an die aufrufende Funktion übergibt.

Diese wird das Ergebnis vom EAX-Register übernehmen.

1.4.2 ARM

Auf der ARM-Plattform gibt es einige kleine Unterschiede:

Listing 1.10: Optimierender Keil 6/2013 (ARM Modus) Assembler-Ausgabe

```
f       PROC
        MOV      r0,#0x7b ; 123
        BX       lr
        ENDP
```

ARM nutzt das Register `R0` für das Speichern des Rückgabewerts der Funktion. Also wird in diesem Beispiel der Wert 123 dorthin kopiert.

Die Rücksprungadresse wird nicht auf dem lokalen Stack sondern im Link-Register gespeichert. Die Anweisung `BX LR` führt dazu, dass die Ausführung an dieser Stelle fortgeführt wird. In diesem Fall wird also die Kontrolle wieder an die aufrufende Funktion übergeben.

Erwähnenswert ist der irreführende Name der `MOV`-Anweisung sowohl beim x86- als auch ARM-Befehlssatz: die Daten werden nicht *verschoben* sondern *kopiert*.

1.4.3 MIPS

Es gibt zwei verschiedene Konventionen bei der Benamung von Registern in der MIPS-Welt: mit einer Nummer (von $0 bis $31) oder mit einem Pseudonamen ($V0, $A0, usw.).

GCC benamt in der Ausgabe die Register mit Nummern:

Listing 1.11: Optimierender GCC 4.4.5 (Assemblercode)

```
        j       $31
        li      $2,123                  # 0x7b
```

…während IDA Pseudonamen verwendet:

Listing 1.12: Optimierender GCC 4.4.5 (IDA)

```
              jr      $ra
              li      $v0, 0x7B
```

Das $2 (oder $V0)-Register wird zum Speichern des Rückgabewerts genutzt. LI steht für ”Load Immediate” und ist das MIPS-Äquivalent zu MOV.

Die anderen Anweisungen sind Sprungbefehle (J oder JR), die die Ausführung wieder an die aufrufende Funktion übergeben, indem an die Adresse gesprungen wird die im $31 (oder $RA)-Register.

Dieses Register ist analog zum LR in der ARM-Architektur.

Möglicherweise wundert man sich warum die Positionen der Lade- (LI) und Sprunganweisung (J or JR) vertauscht sind. Dies geschieht durch ein RISC-Feature das ”branch delay slot” genannt wird.

Die Begründung liegt in der Eigenart einiger RISC-Befehlssets die hier jedoch nicht so wichtig ist. Man sollte aber im Hinterkopf behalten, dass bei MIPS die Anweisung *nach* dem Sprungbefehl noch *vor* dieser ausgeführt wird.

Als Konsequenz sind Verzweigungsbefehle immer mit der Anweisung vertauscht, die zuvor ausgeführt werden muss.

Anmerkungen zu MIPS-Anweisungen / Registernamen

Register- und Anweisungsnamen sind in der MIPS-Welt traditionellerweise in Kleinbuchstaben geschrieben. Aus Gründen der Einheitlichkeit wird in diesem Buch jedoch die Großschreibung bevorzugt.

1.5 Hallo, Welt!

Beginnen wir mit dem berühmten Beispiel aus dem Buch [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

```
#include <stdio.h>

int main()
```

```
{
    printf("hello, world\n");
    return 0;
}
```

1.5.1 x86

MSVC

Das Beispiel wird jezt in MSVC 2010 kompiliert:

```
cl 1.cpp /Fa1.asm
```

(Die /Fa-Option weist den Compiler an, Assembler-Code auszugeben.)

Listing 1.13: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC erstellt Assembler-Code im Intel-Syntax. Der Unterschied zum AT&T-Syntax wird später in 1.5.1 on page 15 behandelt.

Der Compiler generiert die Datei 1.obj, die anschließend zu 1.exe gelinkt wird. In diesem Fall besteht die Datei aus zwei Segmenten: CONST (für konstante Daten) und _TEXT (für Quellcode).

Die Zeichenkette hello, world hat in C/C++ den Typ const char[][Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], aber keinen eigenen Bezeichner. Da der Compiler jedoch irgendwie auf diese Zeichenkette zugreifen muss, definiert er den internen Namen $SG3830.

Aus diesem Grund kann das Beispiel auch wie folgt geschrieben werden:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";
```

```
int main()
{
    printf($SG3830);
    return 0;
}
```

Nochmal zurück zum Assembler-Listing: wie man sehen kann ist die Zeichenkette gemäß dem C/C++-Standard mit einem 0-Byte abgeschlossen. Mehr über C/C++-Zeichenketten ist im Abschnitt 5.4.1 on page 550 zu finden.

In dem Code-Segment _TEXT ist lediglich eine Funktion: `main()`. Diese startet mit einem Prolog-Teil und endet mit einem Epilog-Teil (wie fast alle Funktionen) [14].

Nach dem Funktions-Prolog ist der Aufruf der `printf()`-Funktion zu sehen:
`CALL _printf`. Vor dem Aufruf wird die Adresse der Zeichenkette (oder ein Zeiger darauf) mit dem Inhalt unserer Begrüßung auf dem Stack gespeichert. Dies geschieht durch die PUSH-Anweisung.

Wenn `printf()` die Ausführung wieder an `main()` übergibt, befindet sich die Adresse der Zeichenkette (oder ein Zeiger darauf) immer noch auf dem Stack. Da diese jedoch nicht mehr benötigt wird, muss der Stapel-Zeiger (das ESP-Register) korrigiert werden.

`ADD ESP, 4` bedeutet, dass der Wert 4 zu dem ESP-Rregister-Wert addiert wird.

Warum 4? Da dies ein 32-Bit-Programm ist, werden exakt 4 Byte benötigt um Adressen auf dem Stack abzulegen. Wenn dies x64-Code wäre, würden 8 Byte benötigt. `ADD ESP, 4` ist quasi gleichbedeutend mit `POP Register` jedoch ohne die Verwendung von Registern[15].

Aus dem gleichen Grund generieren einige Compiler (wie der Intel C++-Compiler) die Anweisung `POP ECX` anstatt `ADD` (dieses Muster kann zum Beispiel im Oracle RDBMS-Code gefunden werden, da dieser mit dem Intel-Compiler erstellt wurde). Diese Anweisung hat nahezu den gleichen Effekt, nur dass die Inhalte des ECX-Registers überschrieben werden. Der Intel C++-Compiler nutzt `POP ECX` vermutlich, da der OpCode für diese Anweisung kürzer ist als `ADD ESP, x` (1 Byte für `POP` und 3 Byte für `ADD`),

Nachfolgend ein Beispiel unter der Verwendung von `POP` anstatt `ADD` aus Oracle RDBMS:

Listing 1.14: Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A                 push    ebx
.text:0800029B                 call    qksfroChild
.text:080002A0                 pop     ecx
```

Nachdem `printf()` aufgerufen wurde, enthält der Original-C/C++-Code die Anweisung `return 0` als Rückgabewert der `main()`-Funktion.

In dem hier gezeigten Code ist dies durch die Anweisung `XOR EAX, EAX` realisiert.

[14] Mehr darüber in dem Abschnitt über Funktions-Prologe und -Epiloge (1.6 on page 39).
[15] Statusregister der CPU können sich jedoch ändern

XOR ist lediglich ein „exklusives Oder“[16] aber der Compiler nutzt dies oft anstatt MOV EAX, 0—auch hier wieder aufgrund des leicht kürzeren OpCodes (2 Byte für XOR und 5 Byte für MOV).

Einige Compiler erzeugen SUB EAX, EAX, was *Subtrahiere den Wert in* EAX *vom Wert in* EAX bedeutet. In jedem Fall erzeugt dies einen Wert von Null.

Die letzte Anweisung RET gibt die Ausführungskontrolle wieder an die aufrufende Funktion caller. Üblicherweise ist dies C/C++ CRT[17]-Code welcher wiederum die Kontrolle an das Betriebssystem (BS) übergibt.

GCC

Als nächstes wird der gleiche C/C++-Code mit GCC 4.4.1 unter Linux kompiliert: gcc 1.c -o 1. Mithilfe des IDA-Disassemblers wird untersucht, wie die main()-Funktion erzeugt wurde. IDA nutzt, genau wie MSVX den Intel-Syntax[18].

Listing 1.15: Code in IDA

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world\n"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Das Ergebnis ist fast das gleiche. Die Adresse der hello, world-Zeichenkette (im Daten-Segment) wird zunächst in das EAX-Register geladen und anschließend auf dem Stack gesichert.
Zusätzlich beinhaltet der Funktions-Prolog AND ESP, 0FFFFFFF0h —diese Anweisung richtet den ESP-Register-Wert an eine 16-Byte-Grenze aus. Dies führt dazu, dass alle Werte im Stack auf die gleiche Weise ausgerichtet sind. Die CPU kann Anweisungen schneller ausführen, wenn die zu verarbeitenden Daten auf einer an 4- oder 16-Byte-Grenzen ausgerichteten Adresse liegen.

SUB ESP, 10h reserviert 16 Byte auf dem Stack, auch wenn - wie später gezeigt wird - nur 4 Byte benötigt werden.

Der Grund liegt darin, dass auch die Größe des Stacks an eine 16-Byte-Grenze ausgerichtet ist.

[16] wikipedia
[17] C Runtime library
[18] GCC kann Assembler-Ausgaben im Intel-Syntax erzeugen mit der Options -S -masm=intel.

Die Adresse der Zeichenkette (oder ein Zeiger darauf) wird anschließend direkt ohne die PUSH-Anweisung auf dem Stack gespeichert. ITvar_10 —ist eine lokale Variable und ein Argument für `printf()`. Mehr dazu später.

Anschließend wird die `printf()`-Funktion aufgerufen.

Anders als MSVC erzeugt GCC ohne Optimierung Die Anweisung `MOV EAX, 0` anstatt des kürzeren OpCodes.

Die letzte Anweisung LEAVE ist ein Äquivalent zu der Kombination aus `MOV ESP, EBP` und `POP EBP`. Mit anderen Worten: diese Anweisung setzt den Stapel-Zeiger (ESP) zurück und stellt die initalen Werte des EBP-Registers wieder her. Dies ist notwendig weil die Registerwerte (ESP und EBP) zu Beginn der Funktion (durch `MOV EBP, ESP` / `AND ESP, …`).

GCC: AT&T Syntax

Im nächsten Beispiel ist sichtbar, wie dies im AT%T-Syntax dargestellt werden kann. Dieser Syntax ist sehr viel populärer in der UNIX-Welt.

Listing 1.16: Das Beispiel kompiliert mit GCC 4.7.3

```
gcc -S 1_1.c
```

Das Ergebnis ist wie folgt:

Listing 1.17: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
```

```
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

Der Quellcode beinhaltet Makros (beginnend mit einem Punkt), die hier aber nicht von Belang sind.

An dieser Stelle werden aus Gründen der Übersichtlichkeit alle Makros au0er *.string* ignoriert. Letzeres kodiert eine Null-terminierte Zeichenkette, die einem C-String entspricht.

Die resultierende Ausgabe ist diese [19]:

Listing 1.18: GCC 4.7.3

```
.LC0:
        .string "hello, world\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Einige der Hauptunterschiede zwischen Intel und AT&T-Syntax sin:

- Quell- und Zieloperanden sind in umgekehrter Reihenfolge angegeben.

  Im Intel-Syntax: <Anweisung> <Ziel-Operand> <Quell-Operand>.

  Im AT&T-Syntax: <Anweisung> <Quell-Operand> <Ziel-Operand>.

  Hier ist eine einfache Möglichkeit um sich den Unterschied zu merken: Beim Umgang mit dem Intel-Syntax, kann man sich ein Gleichheitszeichen (=) zwischen den Operanden vorstellen und beim AT&T-Syntax einen Pfeil nach rechts (→) [20].
- AT&T: Vor einem Register-Namen muss ein Prozentzeichen (%) und vor Zahlen ein Dollarzeichen ($) stehen. Statt eckigen werden runde Klammern genutzt.
- AT&T: An eine Anweisung ist ein Suffix angehängt, der die Operandengröße angibt:
  - q — quad (64 bits)
  - l — long (32 bits)
  - w — word (16 bits)

[19]Um die „unnötigen“ Makros zu unterdrücen kann die GCC-Option *-fno-asynchronous-unwind-tables* genutzt werden

[20]Einige C-Standard-Funktionen (z.B. memcpy(), strcpy()) sind die Parameter ebenfalls wie im Intel-Syntax aufgelistet: erst der Zeiger zum Ziel, dann der Zeiger auf die Speicher-Quelle)

- b — byte (8 bits)

Nochmals zu dem kompilierten Ergebnis: Dieses ist identisch mit der Anzeige in IDA, jedoch mit einem kleinen Unterschied: 0FFFFFFF0h wird als $-16 angezeigt. Der eigentliche Wert ist der selbe: 16 im Dezimalsystem ist 0x10 im Hexadezimalsystem. Für 32-Bit-Datentypen ist -0x10 identisch mit 0xFFFFFFF0.

Eine weitere Sache: der Rückgabewert ist mittels MOV auf Null gesetzt, nicht mit XOR. MOV läd lediglich einen Wert in ein Register. Der Name ist irreführend, da die Daten nicht verschoben, sondern kopiert werden. In anderen Architekturen ist wird dieser Befehl „LOAD“ oder „STORE“ oder ähnlich genannt.

String-Patching (Win32)

Man kann die Zeichenkette ”hello, world” in der ausführbaren Datei mit Hiew finden:

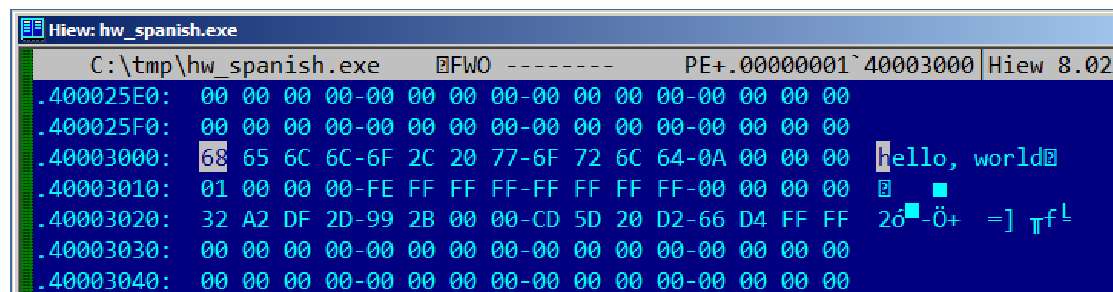


Abbildung 1.1: Hiew

Man kann jetzt versuchen die Meldung ins Spanische zu übersetzen:

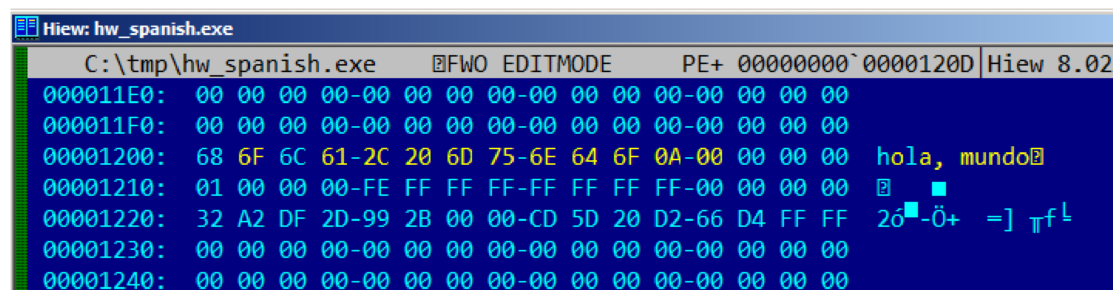


Abbildung 1.2: Hiew

Die spanische Version ist ein Byte kürzer als die englische, als muss am Ende ein 0x0A-Byte (\n) und ein Null-Byte eingefügt werden.

Es funktioniert.

Was wenn eine längere Nachricht eingefügt werden soll? Hinter dem originalen englischen Text befinden sich einige Nullbytes. Es ist schwierig zu sagen, ob diese überschrieben werden dürfen: es ist möglich, dass die zum Beispiel in dem CRT-Code genutz werden. Vielleicht aber auch nicht. Wie dem auch sei: diese Daten sollten nur überschrieben werden, wenn wirklich klar ist was man tut.

String-Patching (Linux x64)

Nachfolgend wird der Patch einer ausführbaren Datei unter einem 64 Bit-Linux mit rada.re gezeigt:

Listing 1.19: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
 -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x004005c4  6865 6c6c 6f2c 2077 6f72 6c64 0000 0000  hello, world....
0x004005d4  011b 033b 3000 0000 0500 0000 1cfe ffff  ...;0...........
0x004005e4  7c00 0000 5cfe ffff 4c00 0000 52ff ffff  |...\...L...R...
0x004005f4  a400 0000 6cff ffff c400 0000 dcff ffff  ....l...........
0x00400604  0c01 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400614  0178 1001 1b0c 0708 9001 0710 1400 0000  .x..............
0x00400624  1c00 0000 08fe ffff 2a00 0000 0000 0000  ........*.......
0x00400634  0000 0000 1400 0000 0000 0000 017a 5200  .............zR.
0x00400644  0178 1001 1b0c 0708 9001 0000 2400 0000  .x..........$...
0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046  ........0......F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422  ..J..w...?.;*3$"
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff  ........D.......
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07  .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff  ....D...d.......
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345  e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e  . ..B.(..H.0..H.

[0x004005c4]> oo+
File a.out reopened in read-write mode

[0x004005c4]> w hola, mundo\x00

[0x004005c4]> q

dennis@bigbox ~/tmp % ./a.out
hola, mundo
```

Was hier passiert ist folgendes: suchen von „hello“ mit dem /-Kommando, dann Setzen des *cursor* (oder *seek* im rada.re-Wording) an diese Adresse. Um sicher zu gehen, dass die richtige Stelle gesetzt ist, kann mit px der Datenblock ausgegeben werden. oo+ versetzt rada.re in den *Lese-Schreibe*-Modus. w schreibt einen ASCII string an die aktuelle Adresse. Hinweis: \00 am Ende ist das Null-Byte. q beendet rada.re.

Software-*Lokalisation* zu MS-DOS-Zeiten

Der hier beschriebene Weg war in den 1980ern und 1990ern sehr vebreitet, um MS-DOS-Progamme in die russische Sprache zu übersetzen. Russische Wörter und Sätze sind in der Regel etwas länger als ihre englischen Gegenstücke, was der Grund ist, dass viele *lokalisierte* Programme eine Menge seltsamer Akronyme und Abkürzungen haben.

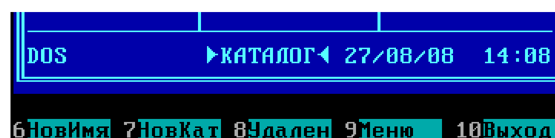


Abbildung 1.3: *Lokalisierter* Norton Commander 5.51

Möglicherweise passierte dies in der Zeit auch in anderen Sprachen.

In Delphi müssen die Längen der Zeichenketten falls nötig korrigiert werden.

1.5.2 x86-64

MSVC: x86-64

Hier das gleiche Beispiel mit der 64-Bit-Variante von MSVC kompiliert:

Listing 1.20: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2989
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

In x86-64 wurden alle Regeister auf 64-Bit erweitert und die Registernamen mit einem R-Prefix versehen. Um den Stack weniger oft zu nutzen (also um auf externen Speicher / Cache selterner zuzugreifen), existiert ein verbreiteter Weg um Funktionsargumente per Register (*fastcall*) 6.1.3 on page 582 zu übergeben. Das heißt ein Teil der Funktionsargumente wird in Registern übergeben, der Rest—über den Stack. In

Win64 werden vier Funktionsargumente in den Registern RCX, RDX, R8 und R9 übergeben. Das ist was hier sichtbar ist: der Zeiger zu der Zeichenkette für `printf()` ist jetzt nicht im Stack übergeben sondern im RCX-Register. Die Zeiger sind nun 64-Bit breit, also werden sie in den 64-Bit-Registern übergeben (die jetz den `R-`Prefix haben). Aus Gründen der Rückwärtskompatibilität ist es aber immer noch möglich mit dem `E-`Prefix auf 32-Bit-Teile zuzugrifen. Nachfolgend, der Aufbau der RAX/EAX/AX/AL-Register in x86-64:

Byte-Nummer:							
7	6	5	4	3	2	1	0
RAX^{x64}							
				EAX			
						AX	
						AH	AL

Die `main()`-Funktion gibt einen Wert vom Typ *int* zurück, der in C/C++ aus Gründen der Kompatibilität und Portabilität immernoch 32 Bit breit ist. Daher wird am Ende der Funktion das EAX-Register auf Null gesetzt (das heißt der 32-Bit-Part des Registers) anstatt RAX. Auf dem lokalen Stack sind zusätzliche 40 Byte reserviert. Dieser Bereich wird „shadow space“ genannt und wird in Abschnitt 1.10.2 on page 113 noch genauer betrachtet.

GCC: x86-64

Nachfolgend das Beispiel unter einem 64 Bit-Linux-System mit GCC kompoliert:

Listing 1.21: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
        xor     eax, eax  ; Anzahl der uebergebenen Register
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Eine Methode im Funktionsargumente in Registern zu übergeben, wird auch in Linux, *BSD und Mac OS X genutzt und heißt [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [21].

Die ersten sechs Argumente sind in den Registern RDI, RSI, RDX, RCX,R8 und R9 übergeben und der Rest—über den Stack.

Der Zeiger zu der Zeichenkette ist in `EDI` (also, dem 32-Bit-Teil) gesichert. Warum wird nicht der 64-Bit-Teil RDI genutz?

[21] Auch verfügbar als `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`

Es ist wichtig sich zu vergegenwertigen, dass alle MOV-Anweisungen im 64-Bit-Modus, die etwas in den niederwertigen 32-Bit-Teil eines Registers schreiben, auch den höherwertigen 32-Bit-Teil des Registers löschen (siehe Intel-Handbücher: 11.1.4 on page 677).
Die Anweisung MOV EAX, 011223344h schreibt also den richtigen Wert in RAX, weil die höherwetigen Bits auf Null gesetzt werden.

In der Objekt-Datei (.o) eines Kompilats sind ebenfalls alles OpCodes der verwendeten Anweisungen zu sehen. [22]:

Listing 1.22: GCC 4.4.6 x64

```
.text:00000000004004D0                          main  proc near
.text:00000000004004D0 48 83 EC 08              sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00           mov     edi, offset format ; "hello,
    world\n"
.text:00000000004004D9 31 C0                    xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF           call    _printf
.text:00000000004004E0 31 C0                    xor     eax, eax
.text:00000000004004E2 48 83 C4 08              add     rsp, 8
.text:00000000004004E6 C3                       retn
.text:00000000004004E6                          main  endp
```

Wie man sehen kann verändert die Anweisung zum Schreiben in EDI an der Adresse 0x4004D4 fünf Byte. Dieselbe Anweisung die einen 64-Bit-Wert in RDI schreibt, verändert 7 Bytes. Offenstichtlich versucht GCC etwas Speicherplatz zu sparen. Nebenbei ist es sicher, dass das Datensegment, welches die Zeichenkette entählt niemals an Adressen höher 4GiB reserviert wird.

Es ist auch erkennbar, dass das EAX-Register vor dem Aufruf von printf() zurückgesetzt wurde. Dies geschieht, aufgrund der Konvention in der oben genannten **ABI!**[23], dass in *NIX-Systemen auf x86-64-Architektur die Anzahl der genutzten Vektor-Register in EAX übergeben wird.

Adress-Patching (Win64)

Wenn das Beispiel in MSCV2013 mit der Option /MD kompiliert wird (was zu einer kleineren ausführbaren Datei durch das linken mit MSVCR*.DLL führt), kommt zuerst die main()-Funktion und kann einfach gefunden werden:

[22] Dies muss aktiviert werden: **Optionen** → **Disassembly** → **Number of opcode bytes**
[23] **ABI!**

```
Hiew: hw2.exe
   C:\tmp\hw2.exe              FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
00000400: 4883EC28                      sub         rsp,028 ;'('
00000404: 488D0DF51F0000                lea         rcx,[000002400]
0000040B: FF15D7100000                  call        q,[0000014E8]
00000411: 33C0                          xor         eax,eax
00000413: 4883C428                      add         rsp,028 ;'('
00000417: C3                            retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
00000418: 4883EC28                      sub         rsp,028 ;'('
0000041C: B84D5A0000                    mov         eax,000005A4D ;'  ZM'
00000421: 663905D8EFFFFF                cmp         [-000000C00],ax
00000428: 7404                          jz          00000042E
000004
000004    lea         rcx,[0000000000002401]
000004
000004            CommandSelect: Off
000004
0000043F: 813850450000                  cmp         d,[rax],000004550 ;'  EP'
00000445: 75E3                          jnz         00000042A
00000447: B90B020000                    mov         ecx,00000020B
0000044C: 66394818                      cmp         [rax][018],cx
00000450: 75D8                          jnz         00000042A
00000452: 33C9                          xor         ecx,ecx
00000454: 83B8840000000E                cmp         d,[rax][000000084],00E
0000045B: 7609                          jbe         000000466
0000045D: 3988F8000000                  cmp         [rax][0000000F8],ecx
1Help  2      3      4Select 5      6      7      8      9      10     11
```

Abbildung 1.4: Hiew

Als Experiment kann die Adresse des Pointers um 1 Inkrement werden:

```
Hiew: hw2.exe
   C:\tmp\hw2.exe          FUO -------- a64 PE+.00000001`4000100B Hiew 8.02 (c)SEN
.40001000: 4883EC28                  sub         rsp,028 ;'('
.40001004: 488D0DF61F0000            lea         rcx,[00000001`40003001] ;'ello, w
.4000100B: FF15D7100000              call        printf
.40001011: 33C0                      xor         eax,eax
.40001013: 4883C428                  add         rsp,028 ;'('
.40001017: C3                        retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.40001018: 4883EC28                  sub         rsp,028 ;'('
.4000101C: B84D5A0000                mov         eax,000005A4D ;'  ZM'
.40001021: 663905D8EFFFFF            cmp         [00000001`40000000],ax
.40001028: 7404                      jz          .00000001`4000102E --2
.4000102A: 33C9                     5xor         ecx,ecx
.4000102C: EB38                      jmps        .00000001`40001066 --3
.4000102E: 48630507F0FFFF           2movsxd      rax,d,[00000001`4000003C] --4
.40001035: 488D0DC4EFFFFF            lea         rcx,[00000001`40000000]
.4000103C: 4803C1                    add         rax,rcx
.4000103F: 813850450000              cmp         d,[rax],000004550 ;'  EP'
.40001045: 75E3                      jnz         .00000001`4000102A --5
.40001047: B90B020000                mov         ecx,00000020B
.4000104C: 66394818                  cmp         [rax][018],cx
.40001050: 75D8                      jnz         .00000001`4000102A --5
.40001052: 33C9                      xor         ecx,ecx
.40001054: 83B8840000000E            cmp         d,[rax][000000084],00E
.4000105B: 7609                      jbe         .00000001`40001066 --3
.4000105D: 3988F8000000              cmp         [rax][0000000F8],ecx
1Help  2PutBlk 3Edit  4Mode  5Goto  6Refer  7Search 8Header 9Files 10Quit  11Hem
```

Abbildung 1.5: Hiew

Hiew zeigt „ello, world“ als Zeichenkette und beim Ausführen der gepatchten Datei wird eben dieser Text ausgegeben.

Aussuchen einer anderen Zeichenkette einer Binärdatei (Linux x64)

Die Binärdatei die beim Kompilieren des Beispiels mit GCC 5.4.0 unter Linux x64 entsteht, beinhaltet noch viele andere Zeichenketten: die meisten sind importierte Funktions- und Bibliotheksnamen.

Mit objdump können die Inhalte aller Sektionen der kompilierten Datei ausgegeben werden:

```
$ objdump -s a.out

a.out:     file format elf64-x86-64

Contents of section .interp:
 400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 400248 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.ABI-tag:
 400254 04000000 10000000 01000000 474e5500  ............GNU.
 400264 00000000 02000000 06000000 20000000  ............ ...
```

```
Contents of section .note.gnu.build-id:
 400274 04000000 14000000 03000000 474e5500  ............GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.......^...
 400294 cf3f7ae4                             .?z.

...
```

Es ist kein Problem die Adresse der Zeichenkette „/lib64/ld-linux-x86-64.so.2“ an `printf()` zu übergeben:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

Schwer zu glauben, aber dieser Code gibt die erwähnte Zeichenkette aus.

Beim Ändern der Adresse zu `0x400260` wird die Zeichenkette „GNU“ ausgegeben. Diese Adresse gilt für die hier verwendete GCC-Version, Toolkonfiguration und so weiter. Auf anderen Systemen kann die ausfühbare Datei leicht unterschiedlich sein, was auch die Adressen verändern kann. Auch das Hinzufügen und Entfernen von Quellcode kann Adressen vor- und zurückschieben.

1.5.3 ARM

Für die Experimente mit ARM-Prozessoren wurden verschiedene Compiler genutzt:

- Verbreitet im Embedded-Bereich: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE mit dem LLVM-GCC 4.2-Compiler [24].
- GCC 4.9 (Linaro) (für ARM64), verfügbar als Win32-Executable unter `http://www.linaro.org/projects/armv8/`.

Wenn nicht anders angegeben wird immer der 32-Bit ARM-Code (inklusive Thumb und Thumb-2-Mode) genutzt. Wenn von 64-Bit ARM die Rede ist, dann wird ARM64 geschrieben.

Nicht optimierender Keil 6/2013 (ARM Modus)

Beginnen wir mit dem Kompilieren des Beispiels mit Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Der *armcc*-Compiler erstellt Assembler-Quelltext im Intel-Syntax, hat aber High-Level-Makros bezüglich der ARM-Prozessoren[25]. Es ist hier wichtig die „richtigen“ Anweisungen zu sehen, deswegen ist hier das Ergebnis mit IDA kompiliert.

[24] Tatsächlich nutzt Apple Xcode 4.6.3 GCC als Front-End-Compiler und LLVM Code Generator
[25] d.h. der ARM-Mode hat keine PUSH/POP-Anweisungen

Listing 1.23: Nicht optimierender Keil 6/2013 (ARM Modus) IDA

```
.text:00000000              main
.text:00000000 10 40 2D E9    STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2    ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB    BL      __2printf
.text:0000000C 00 00 A0 E3    MOV     R0, #0
.text:00000010 10 80 BD E8    LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+4
```

Im ersten Beispiel ist zu erkennen, dass jede Anweisung 4 Byte groß ist. Tatsächlich wurde der Code für den ARM- und nicht den Thumb-Mode erstellt.

Die erste Anweisung, `STMFD SP!, {R4,LR}`[26], arbeitet wie eine x86-PUSH-Anweisung um die Werte der beiden Register (R4 and LR) auf den Stack zu legen.

Die Ausgabe des *armcc*-Compilers, zeigt, aus Gründen der Einfachheit, die `PUSH {r4,lr}`-Anweisung. Dies ist nicht vollständig präzise. Die PUSH-Anweisung ist nur im Thumb-Mode verfügbar. Um die Dinge nicht zu verwirrend zu machen, wird der Code in IDA kompiliert.

Die Anweisung dekrementiert zunächst den **SP!**[28], so dass er auf den Bereich im Stack zeigt, der für neue Einträge frei ist. Anschließend werden die Werte der Register R4 und LR an der Adresse gespeichert auf den der (modifizierte) **SP!** zeigt.

Diese Anweisungen (wie PUSH im Thumb-Mode) ist in der Lage mehrere Register-Werte auf einmal zu speichern, was sehr nützlich sein kann. Übrigens: in x86 gibt es dazu kein Äquivalent. Außerdem ist erwähnenswert, dass die STMFD-Anweisung eine Generalisierung der PUSH-Anweisung (ohne deren Eigenschaften) ist, weil sie auf jedes Register angewandt werden kann und nicht nur auf **SP!**. Mit anderen Worten kann STMFD genutzt werden um eine Reihen von Registern an einer angegebenen Speicher-Adresse zu sichern.

Die `ADR R0, aHelloWorld`-Anweisung addiert oder subtrahiert den Wert im **PC!**[29]-Register zum Offset an dem die `hello, world`-Zeichenkette ist. Man kann sich nun fragen, wie das PC-Register hier genutzt wird. Dies wird „positionsabhängiger Code"[30] genannt.

Code dieser Art kann an nicht-festen Adressen im Speicher ausgeführt werden. Mit anderen Worten: dies ist **PC!**-relative Adressierung. Die ADR-Anweisung berücksichtigt den Unterschied zwischen der Adresse dieser Anweisung und der Adresse an dem die Zeichenkette gespeichert ist. Der Unterschied (Offset) ist immer gleich, egal an welcher Adresse der Code vom BS geladen wurden. Dementsprechend ist alles was gemacht werden muss, die Adresse der aktuellen Anweisung (vom **PC!**) zu addieren um die absolute Speicheradresse der Zeichenkette zu bekommen.

[26] STMFD[27]

[28] **SP!**

[29] **PC!**

[30] mehr darüber in der entsprechenden Sektion (6.4.1 on page 600)

`BL __2printf`[31]-Anweisung ruft die `printf()`-Funktion auf. Die Anweisung funktioniert wie folgt:

- Speichere die Adresse hinter der BL-Anweisung (`0xC`) in LR;
- anschließend wird übergebe die Kontrolle an `printf()` indem dessen Adresse ins **PC!**-Register geschrieben wird.

Wenn `printf()` die Ausführung beendet, müssen Informationen vorliegen, wo die Ausführung weitergehen soll. Das ist der Grund warum jede Funktion die Kontrolle an die Adresse, gespeichert im LR-Register übergibt.

Dies ist ein Unterschied zwischen einem „reinem" RISC-Prozessor wie ARM und CISC[32]-Prozessoren wie x86, bei denen die Rücksprungadresse in der Regel auf dem Stack gespeichert wird. Mehr dazu ist im nächsten Abschnitt zu lesen (1.7 on page 40).

Übrigens eine absolute 32-Bit-Adresse oder -Offset kann nicht in einer 32-Bit-BL-Anweisung kodiert werden, weil diese nur für 24 Bit Platz bietet. Wie bereits erwähnt haben alle ARM-Mode-Anweisungen eine Größe von 4 Byte (32 Bit). Aus diesem Grund können diese nur an 4-Byte-Grenzen des Speichers platziert werden. Dies heißt auch, das die letzten zwei Bit der Anweisungsadresse (die immer Null sind) entfallen können. Zusammenfassend, stehen 26 Bit für die Offset-Kodierung zur Verfügung. Dies ist genug für $current_PC \pm \approx 32M$.

Als nächstes schreibt die Anweisung `MOV R0, #0`[33] lediglich 0 in das `R0`-Register weil der Rückgabewert hier gespeichert wird und die gezeigte C-Funktion 0 als Argument für die return-Anweisung hat.

Die letzte Anweisung `LDMFD SP!, R4,PC`[34] lädt die Werte nacheinander vom Stack (oder eine andere Speicheradresse) um sie in die Register R4 und **PC!** zu sichern. Außerdem wird der Stack Pointer **SP!** inkrementiert. Hier arbeitet der Befehl wie `POP`.

Die erste Anweisung `STMFD` sichert das Register-Paar R4 und LR auf dem Stack, jedoch werden R4 und **PC!** während der Ausführung von `LDMFD` *wiederhergestellt*.

Wie bereits bekannt, wird die Adresse die nach der Ausführung einer Funktion angesprungen wird in dem LR-Register gesichert. Die allererste Anweisung sichert diese Wert auf dem Stack weil das gleiche Register von der `main()`-Funktion genutzt wird, wenn `printf()` aufgerufen wird. Am Ende der Funktion kann dieser Wert direkt in das **PC!**-Register geschrieben werden und so die Ausführung an der Stelle fortgesetzt werden an der die Funktion aufgerufen wurde.

Da `main()` in der Regel die erste Funktion in C/C++ ist, wird die Kontrolle an das BS oder einen Punkt in der CRT übergeben.

All dies erlaubt das Auslassen der `BX LR`-Anweisung am Ende der Funktion.

`DCB` ist eine Assemblerdirektive die ein Array von Bytes oder ASCII anlegt, ähnlich der DB-Direktive in der x86-Assembler-Sprache.

[31]Branch with Link
[32]Complex Instruction Set Computing
[33]das heißt MOVe
[34]LDMFD[35] ist eine inverse Anweisung von STMFD

Nicht optimierender Keil 6/2013 (Thumb Modus)

Nachfolgend das gleiche Beispiel mit dem Keil-Compiler im Thumb-Mode erstellt:

```
armcc.exe --thumb --c90 -O0 1.c
```

In IDA wird folgende Ausgabe erzeugt:

Listing 1.24: Nicht optimierender Keil 6/2013 (Thumb Modus) + IDA

```
.text:00000000              main
.text:00000000 10 B5           PUSH    {R4,LR}
.text:00000002 C0 A0           ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9     BL      __2printf
.text:00000008 00 20           MOVS    R0, #0
.text:0000000A 10 BD           POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ; DATA XREF:
    main+2
```

Leicht zu erkennen sind die 2-Byte (16 Bit) OpCodes, die wie bereits erwähnt Thumb-Anweisungen sind. Die BL-Anweisung besteht aus zwei 16-Bit-Anweisungen, weil es für die `printf()`-Funktion unmöglich ist einen Offset zu laden, wenn der kleine Speicherbereich in einem 16-Bit-Opcode genutzt wird. Aus diesem Grund lädt die erste 16-Bit-Anweisung die höherwertigen 10 Bit des Offsets und die zweite Anweisung die niederwertigen 11 Bit.

Wie erwähnt haben alle Anweisungen im Thumb-Mode eine Größe von 2 Byte (16 Bit). Dies bedeutet, dass es unmöglich ist an einer ungeraden Adresse einen Anweisung unterzubringen. Das hat auch zur Folge, dass das letzte Bit der Adresse bei der Kodierung der Anweisungen weggelassen werden kann.

Zusammenfassend kann die BL-Thumb-Anweisung eine Adresse bis $current_PC \pm \approx 2M$ kodieren.

Wie für die anderen Anweisungen in dieser Funktion arbeiten PUSH und POP wie die beschriebenden STMFD/LDMFD, nur dass das **SP!**-Register hier nicht explizit genannt wird. ADR arbeitet genau wie in dem vorherigen Beispiel. MOVS schreibt 0 in das Register R0 um 0 zurückzugeben.

Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Xcode 4.6.3 ohne Optimierung produziert eine Menge redundanten Code, so dass im Folgenden die optimierte Ausgabe gelistet ist bei der die Anzahl der Anweisungen so klein wie möglich ist. Der Compiler-Schalter ist -O3.

Listing 1.25: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
__text:000028C4             _hello_world
__text:000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV             R7, SP
__text:000028D0 00 00 40 E3   MOVT            R0, #0
__text:000028D4 00 00 8F E0   ADD             R0, PC, R0
```

```
__text:000028D8 C3 05 00 EB   BL              _puts
__text:000028DC 00 00 A0 E3   MOV             R0, #0
__text:000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world!",0
```

Die Anweisungen `STMFD` und `LDMFD` sind bereits bekannt.

Die `MOV`-Anweisung schreibt lediglich die Nummer `0x1686` in das Register `R0`. Dies ist der Offset der auf die Zeichenkette „Hello world!“ zeigt.

Das Register R7 (spezifiziert in [*iOS ABI Function Call Guide*, (2010)][36]) ist ein Frame Pointer. Mehr darüber folgt später.

Die `MOVT R0, #0` (MOVe Top)-Anweisung schreibt 0 in die höherwertigen 16 Bit des Registers. Das Problem ist hier, dass die generische `MOV`-Anweisung im ARM-Mode nur die niederwertigen 16 Bit des Registers beschreibt.

Dran denken: alle Opcodes im ARM-Mode sind in der Größe auf 32 Bit begrenzt. Natürlich gilt diese Begrenzung nicht für das Verschieben von Daten zwischen Registern. Aus diesem Grund existiert die zusätzliche Anweisung `MOVT` um in die höherwertigen Bits (von 16 bis einschließlich 31) zu beschreiben. Die Benutzung ist in diesem Fall redundant, weil die Anweisung `MOV R0, #0x1686` darüpber den höherwertigen Teil des Registers zurückgesetzt hat. Dies ist vermutlich ein Mangel des Compilers.

Die Anweisung `ADD R0, PC, R0` addiert den Wert im **PC!** zum Wert im Register `R0` um die absolute Adresse der „Hello world!“-Zeichenkette zu berechnen. Wie bereits bekannt ist dies „positionsabhängiger Code“, so dass diese Korrektur hier unbedingt notwendig ist.

Die `BL`-Anweisung ruft `puts()` anstatt `printf()` auf.

LLVM ersetzt den ersten `printf()`-Aufruf mit `puts()`. In der Tat ist `printf()` mit nur einem Argument identisch mit `puts()`.

Die beiden Funktionen produzieren lediglich das gleiche Ergebnis, weil printf keine Formatkennzeichner, beginnend mit %, enhält. Sollte dies jedoch der Fall sein, wäre die Auswirkung der beiden Funktionen unterschiedlich[37].

Warum hat der Compiler diese Ersetzung durchgeführt? Vermutlich hat dies Vorteile bei der Geschwindigkeit, weil `puts()` schneller ist [38] und lediglich die Zeichen zu stdout übergibt, anstatt jedes Zeichen mit % zu vergleichen.

Als nächstes ist die bekannte Anweisung `MOV R0, #0` zu sehen um das Register `R0` auf 0 zu setzen.

Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

Standardmäßig generiert Xcode 4.6.3 den Thumb-2-Code auf folgende Weise:

[36] Auch verfügbar als http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

[37] Des weiteren benötigt `puts()` kein '\n' für den Zeilenumbruch am Ende der Zeichenkette, weswegen wir dies hier nicht sehen.

[38] ciselant.de/projects/gcc_printf/gcc_printf.html

Listing 1.26: Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
__text:00002B6C                         _hello_world
__text:00002B6C 80 B5          PUSH           {R7,LR}
__text:00002B6E 41 F2 D8 30    MOVW           R0, #0x13D8
__text:00002B72 6F 46          MOV            R7, SP
__text:00002B74 C0 F2 00 00    MOVT.W         R0, #0
__text:00002B78 78 44          ADD            R0, PC
__text:00002B7A 01 F0 38 EA    BLX            _puts
__text:00002B7E 00 20          MOVS           R0, #0
__text:00002B80 80 BD          POP            {R7,PC}

...

__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

Die BL- und BLX-Anweisung im Thumb-Mode ist als Paar von 16-Bit-Anweisungen kodiert. In Thumb-2 sind diese *Ersatz*-Opcodes so erweitert, dass neue Anweisungen hier mit 32 Bit kodiert werden können.

Offensichtlich beginnen die Opcodes der Thumb-2-Anweisungen immer mit `0xFx` oder `0xEx`.

Im IDA-Listing jedoch sind die Bytes der Opcodes vertauscht weil für den ARM-Prozessor die Anweisungen wie folgt kodiert werden: Das letzte Byte kommt zuerst und danach das erste (für Thumb- und Thum-2-Mode) oder für Anweisungen im ARM-Mode kommt das vierte Byte zuerst, dann das dritte, dann das zweite und zum Schluss das erste (aufgrund des unterschiedlichen Endianness).

Die Bytes sind also im IDA-Listing wie folgt angeordnet:

- für ARM und ARM64 Mode: 4-3-2-1;
- für Thumb Mode: 2-1;
- für 16-Bit-Anweisungspaar in Thumb-2 Mode: 2-1-4-3.

Wie zu sehen ist, beginnend die Anweisungen `MOVW`, `MOVT.W` und BLX mit `0xFx`.

Eine der Thumb-2-Anweisungen ist `MOVW R0, #0x13D8` —sie speichert einen 16-Bit-Wert in den niederwertigeren Teil des `R0`-Registers und setzt die höherwertigen Bits auf 0.

Des weiteren funktioniert `MOVT.W R0, #0` genau wie `MOVT` aus dem vorherigen Beispiel, jedoch nur für Thumb-2.

Neben den anderen Unterschieden wird in diesem Fall die BLX-Anweisung anstatt BL genutzt.

Der Unterschied ist, dass, neben dem Speichern von RA[39] in das LR-Register und die Übergabe der Ausführungskontrolle an die `puts()`-Funktion, der Prozessor auch vom Thumb/Thumb-2-Mode in den ARM-Mode (oder zurück) wechselt.

Diese Anweisung ist hier eingefügt weil die Anweisung mit der die Kontrolle abgegeben wird wie folgt aussieht (im ARM-Mode kodiert):

[39]Rücksprungadresse

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

Dies ist im Endeffekt ein Sprung an die Stelle an der die Adresse von `puts()` in der import-Sektion geschriben wird.

Der aufmerksame Leser mag fragen: warum wird `puts()` nicht direkt an der Stelle im Code aufgerufen, an der es benötigt wird? Dies wäre nicht sehr speicherplatzeffizient.

Fast jedes Programm nutzt externe, dynamische Bibliotheken (wie DLL in Windows, .so in *NIX oder.dylib in Mac OS X). Diese Bibliotheken beinhalten häufig genutzte Funktion wie die Standard-C-Funktion `puts()`.

In einer ausführbaren Binärdatei (Windows PE .exe, ELF oder Mach-O) existiert eine import-Sektion. Dies ist eine Liste von Symbolen (Funktionen oder globale Variablen) die, zusammen mit den Namen, von externen Modulen importiert werden.

Der BS-Loader läd alle Module die gebraucht werden und bestimmt die korrekten Adressen von jedem Symbol, während diese in dem primärem Modul aufgelistet werden.

In dem vorliegenden Fall ist *__imp__puts* eine 32-Bit-Variable die vom BS-Loader genutzt wird um die korrekte Adresse der Funktion in der externen Bibliothek zu speichern. Anschließend liest die LDR-Anweisung den 32-Bit-Wert dieser Variable und schreibt ihn in das **PC!**-Register bevor die Ausführkontrolle dorthin übergeben wird.

Um also die Zeit zu reduzieren die der BS-Loader für dieses Vorgehen benötigt, ist es eine gute Idee die Adressen für jedes Symbol einmalig an eine geeignete Stelle zu schreiben.

Daneben wurde bereits erwähnt, dass es unmöglich ist einen 32-Bit-Wert in ein Register zu laden wenn nur eine Anweisung ohne Speicher-Zugriff genutzt wird.

Aus diesem Grund ist die optimale Lösung, eine separate Funktion im ARM-Mode zu allozieren die lediglich die Aufgabe hat die Ausführkontrolle an die dynamische Bibliothek zu übergeben und dann in diese kurze Funktion mit einer Anweisung (so genannte Thunk-Funktion) aus dem Thumb-Code auszuführen.

Übrigens: in dem vorherigen Beispiel (für ARM-Mode kompiliert) wird die Ausführkontrolle durch BL an die gleiche Thunk-Funktion übergeben. Der Prozessor-Modus wird hier jedoch aufgrund des Fehlens eines „X" im Anweisungsnamen nicht gewechselt.

Mehr über Thunk-Funktionen

Thunk-Funktionen sind aufgrund der irrtümlichen Bezeichnung schwierig zu verstehen. Der einfachste Weg ist es sie als Adapter oder Konverter zwischen verschiedenen Anschlüssen aufzufassen. Zum Beispiel wie einen Adapter zwischen einer britischen und einer amerikanischen Steckdose oder andersherum. Thunk-Funktionen werden manchmal auch *Wrapper* genannt.

Hier sind einige weitere Beschreibung dieser Funktionstypen:

> ”Ein Teil der Software um Adressen zur Verfügung zu stellen:” nach P. Z. Ingerman, der 1961 Thunk-Funktionen als Möglichkeit zum Binden von Aktualparametern zu deren formalen Definitionen in Algol-60-Prozedur-Aufrufen. Wenn eine Prozedur mit einem Ausdruck anstatt der formalen Parameter aufgerufen wird, generiert der Compiler eine Thunk-Funktion die den Ausdruck errechnet und die Adresse des Ergebnisses an eine Standard-Stelle speichert.
>
> ...
>
> Microsoft und IBM haben beide in ihrem Intel-basierten System eine ”16-Bit Umgebung” und eine ”32-Bit-Umgebung” definiert. Beide können auf dem selben Computer und demselben Betriebssystem laufen (dank dem was Microsoft „ Windows On Windows“ (WOW) nennt). Sowohl MS als auch IBM haben entschieden, den Vorgang der zwischen 16- und 32-Bit wechselt ”Thunk” zu nennen; für Windows 95 existiert sogar ein Tool THUNK.EXE, das Thunk-Compiler genannt wird.

(The Jargon File)

ARM64

GCC

Das Beispiel wird im Folgenden mit GCC 4.1.8 in ARM64 kompiliert:

Listing 1.27: Nicht optimierender GCC 4.8.1 + objdump

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210a00  ........Hello!..
```

Es gibt keine Thumb- oder Thumb-2-Modes in ARM64, sondern nur ARM, also 32-Bit-Anweisungen. Die Register-Anzahl ist verdoppelt: **??** on page ??. 64-Bit-Register haben einen X-Prefix, 32-Bit-Teile ein W-.

Die STP-Anweisung (*Store Pair*) speichert zwei Register auf dem Stack gleichzeitig: X29 und X30.

Natürlich kann diese Anweisung dieses Registerpaar an einer beliebigen Stelle im

Speicher sichern, aber da hier das **SP!**-Register angegeben ist, wird das Paar auf dem Stack gesichert.

ARM64-Register sind 64 Bit breit, jedes von ihnen ist 8 Byte groß. Dementsprechend werden 16 Byte für das Speichern zweier Register benötigt.

Das Ausrufungszeichen ("!") nach dem Operanden bedeutet, dass zunächst der Wert 16 vom **SP!** subtrahiert werden muss und erst dann die Werte vom Register-Paar auf den Stack geschrieben werden. Dies wird auch *pre-index* genannt. Mehr über den Unterschied von *post-index* und *pre-index* ist im Abschnitt 1.30.2 on page 524 zu finden.

Im Sprachgebrauch des gebräuchlicheren x86, ist die erste Anweisung analog zu den Anweisungen `PUSH X29` und `PUSH X30` zu verstehen. X29 wird als FP[40] in ARM64 genutzt, und X30 als LR, weswegen sie am Anfang der Funktion gesichert und am Ende wiederhergestellt werden.

Die zweite Anweisung kopiert **SP!** in X29 (oder FP) um den Stack Frame vorzubereiten.

`ADRP` und `ADD`-Anweisungen werden genutzt um die Adresse der Zeichenkette „Hello!" in das Register `X0` zu schreiben, da das erste Funktionsargument in an dieser Stelle übergeben wird.

Es gibt in ARM keine Anweisung, die eine große Zahl in einem Register sichern kann, weil die Länge der Anweisungen auf 4 Byte begrenzt ist. Siehe dazu auch 1.30.3 on page 526). Aus diesem Grund müssen mehrere Anweisungen genutzt werden. Die erste (`ADRP`) schreibt die Adresse der 4KiB-Page in der die Zeichenkette sich befindet in das Register `X0`. Die zweite (`ADD`) addiert lediglich den Rest der Adresse. Siehe dazu auch 1.30.4 on page 528.

`0x400000 + 0x648 = 0x400648`, und die Zeichenkette „Hello!" ist im `.rodata` Daten-Segmet an dieser Adresse zu sehen.

`puts()` wird anschließend mit der `BL`-Anweisung aufgerufen. Dies wurde bereits diskutiert: 1.5.3 on page 28.

`MOV` schreibt 0 in `W0`. `W0` sind die niederwertigeren 32 Bit des 64-Bit-Registers `X0`:

Oberer 32-Bit-Teil	Unterer 32-Bit-Teil
X0	
	W0

Das Ergebnis der Funktion wird über `X0` zurückgegeben und `main()` gibt 0 zurück. Dies ist also der Weg wie das Ergebnis vorbereitet wird. Der 32-Bit-Teil wird genutzt,weil der *int*-Datentyp in ARM64 aus Kompatibilitätsgründen, wie in x86-64, 32 Bit breit ist.

Da die Funktion einen 32-Bit *int*-Wert zurück gibt, müssen lediglich die unteren 32 Bits des `X0`-Registers gefüllt werden.

Um dies zu überprüfen wird das Beispiel leicht verändert und neu kompiliert. `main()` soll nun einen 64-Bit-Wert zurück geben:

[40] Frame Pointer

Listing 1.28: main() gibt einen uint64_t-Datentyp zurück

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
        printf ("Hello!\n");
        return 0;
}
```

Das Ergebnis ist das gleiche, allerdings sieht MOV nun wie folgt aus:

Listing 1.29: Nicht optimierender GCC 4.8.1 + objdump

```
  4005a4:       d2800000        mov     x0, #0x0      // #0
```

LDP (*Load Pair*) stellt anschließend die Register X29 und X30 wieder her.

An dieser Stelle steht kein Ausrufungszeichen nach der Anweisung: dies impliziert, dass der Wert zunächst vom Stack gelesen wird und erst dann wird **SP!** um den Wert 16 verringert. Dies wird *post-index* genannt.

Eine neue Anweisung taucht hier in ARM64 auf RET. Diese arbeitet wie BX LR, jedoch wird ein spezielles *Hinweis*-Bit hinzugefügt, welches die CPU darüber informiert, dass dies ein Rücksprung aus einer Funktion ist und kein anderer Sprung, so dass die Ausführung optimiert werden kann.

Aufgrund der Einfachheit dieser Funktion, erstellt der optimierende GCC den gleichen Code.

1.5.4 MIPS

ein Wort über „globale Zeiger“

Ein wichtiges Konzept bei MIPS ist der „globale Zeiger“. Wie bereits bekannt, besteht besteht jede MIPS-Anweisung aus 32 Bit, so dass es nicht möglich ist eine 32-Bit-Anweisung darin unterzubringen: ein Anweisungspaar wird verwendet (wie GCC dies in dem Beispiel zum Laden der Zeichenkettenadresse getan hat).

Es ist jedoch möglich, Daten aus dem Adressbereich $register - 32768...register + 32767$ mit nur einer Anweisung zuladen, weil ein 16 Bit vorzeichenbehafteter Offset in einer einzelnen Anweisung kodiert werden kann. Es können also einige Register zu diesen Zweck alloziert werden und 64KiB-Bereiche für die am häufigsten genutzten Daten Dieses allozierte Register wird „globaler Zeiger“ genannt und zeigt in die Mitte des 64KiB-Bereichs.

Dieser Bereich enthält in der Regel globale Variablen und Adressen von importierten Funktionen wie printf(), weil die GCC-Entwickler entschieden, dass das Laden einiger Funktionsadressen so schnell sein sollte wie eine einzelne Anweisung anstatt zwei. In einer ELF-Datei ist dieser 64KiB-Bereich teils in der Sektion .sbss („small BSS[41]“) für uninitialiserte Daten und teil in .sdata („small data“) für initialisierte Daten zu finden.

[41] Block Started by Symbol

Dies impliziert, dass der Programmierer entscheiden kann, auf welche Daten ein schneller Zugriff (durch das Platzieren in .sdata/.sbss) möglich sein soll. Einige Programmierer „der alten Schule“ erinnern sich vielleicht an das MS-DOS Speichermodell 10.7 on page 672 oder MS-DOS Speicherverwaltungen wie XMS/EMS bei denen der komplette Speicher in 64KiB-Blöcke unterteilt war.

Dieses Konzept ist nicht nur bei MIPS vorhanden. Zumindest der PowerPC nutzt es ebenfalls.

Optimierender GCC

Nachfolgen ein Beispiel welches das Konzept der „globalen Zeiger“ veranschaulichen soll.

Listing 1.30: Optimierender GCC 4.4.5 (Assemblercode)

```
$LC0:
; \000 ist das Nullbyte im Oktalsystem:
        .ascii  "Hello, world!\012\000"
main:
; Funktionsprolog:
; Setze den globalen Zeiger:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
; speichere Ruecksprungadresse auf lokalem Stack:
        sw      $31,28($sp)
; Lade Adresse von puts() function vom glob. Zeiger in $25:
        lw      $25,%call16(puts)($28)
; Lade Adresse der Zeichenkette in $4 ($a0):
        lui     $4,%hi($LC0)
; Springe zu puts(), speichere Ruecksprungadresse im Link Register:
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; Ruecksprungadresse wiederherstellen:
        lw      $31,28($sp)
; Kopiere 0 von $zero zu $v0:
        move    $2,$0
; Springe an Ruecksprungadresse:
        j       $31
; Funktionsepilog:
        addiu   $sp,$sp,32 ; branch delay slot
```

Wie zu sehen wird das $GP-Register im Funktionsprolog so gesetzt, dass auf die Mitte dieses Bereichs gezeigt wird. Das RA-Register wird ebenfalls auf dem lokalen Stack gesichert. Anstelle von `printf()` wird wieder `puts()` aufgerufen. Die Adresse der Funktion `puts()` wird mit der LW-Anweisung („Load Word“) in $25 geladen. Anschließend wird die Adressse der Zeichenkette mit dem Anweisungspaar LUI („Load Upper Immediate“) und ADDIU („Add Immediate Unsigned Word“) in $4 geladen. LUI setzt die oberen 16 Bit des Registers (deswegen „upper“ im Anweisungsnamen) und ADDIU addiert die unteren 16 Bit der Adresse.

ADDIU folgt JALR (zur Erinnerung: *branch delay slots*). Das Register $4 wird auch $A0 genannt und für das Übergeben des ersten Funktionsarguments genutzt[42].

JALR („Jump and Link Register“) springt zu der Adresse die im Register $25 gespeichert ist (Adresse von puts()) und speichert die Adresse der übernächsten Anweisung (LW) in RA. Dies ist sehr ähnlich zu ARM. Eine wichtige Sache ist, dass die Adresse in RA nicht die Adresse der nächsten Anweisung ist (da dies ein *delay slot* ist und vor der Sprunganweisung ausgeführt wird), sondern die Adresse der darauf folgenden Anweisung (nach dem *delay slot*). Da in diesem Fall während der Ausführung von JALR der Wert $PC + 8$ in RA geschrieben wird, ist dies die Adresse der LW-Anweisung nach ADDIU.

LW („Load Word“) in Zeile 20 stellt RA wieder vom lokalen Stack her. Diese Anweisung ist tatsächlich ein Teil des Funktionsepilogs.

MOVE in Zeile 22 kopiert der Wert vom $0 ($ZERO)-Register in $2 ($V0).

MIPS besitzt ein *konstantes* Register, welches immer eine Null beinhaltet. Anscheinend hatten die MIPS-Entwickler die Idee, dass eine Null die beliebteste Konstante in der Programmierung ist, also wird in Zukunft immer das $0-Register genutzt wenn eine Null benötigt wird.

Eine weitere interessante Tatsache in MIPS ist das Fehlen einer Anweisung zum Transferieren von Daten zwischen zwei Registern. Die Anweisung MOVE DST, SRC entspricht jedoch ADD DST, SRC, $ZERO ($DST = SRC + 0$), und bewirkt genau das gleiche. Anscheinend wollten die MIPS-Entwickler eine kompakte Opcode-Tabelle haben. Das bedeutet nicht, dass dies bei jeder MOVE-Anweisung passiert. Sehr wahrscheinlich optimiert die CPU diese Pseudo-Anweisung und die ALE[43] wird niemals genutzt.

J in Zeile 24 springt zu der Adresse in RA, was im Endeffekt einem Sprung aus einer Funktion entspricht. ADDIU nach J wird tatsächlich bevor J ausgeführt (siehe *branch delay slots*) und ist ein Teil des Funktions-Epilogs. Hier ist die Ausgabe, die IDA generiert. Jedes Register hat einen eigenen Pseudo-Namen:

Listing 1.31: Optimierender GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_4           = -4
.text:00000000
; Funktionsprolog:
; GP setzen:
.text:00000000                 lui     $gp, (__gnu_local_gp >> 16)
.text:00000004                 addiu   $sp, -0x20
.text:00000008                 la      $gp, (__gnu_local_gp & 0xFFFF)
; RA auf lokalem Stack sichern:
.text:0000000C                 sw      $ra, 0x20+var_4($sp)
; GP auf lokalem Stack sichern:
; diese Anweisung fehlt aus irgendeinem Grund bei GCC:
.text:00000010                 sw      $gp, 0x20+var_10($sp)
```

[42]Die MIPS-Register-Tabelle ist im Anhang verfügbar **??** on page ??

[43]Arithmetisch-logische Einheit

```
16 ; Lade Adresse von puts() von GP in $t9:
17 .text:00000014                lw      $t9, (puts & 0xFFFF)($gp)
18 ; Adresse der Zeichenkette in $a0 hinterlegen:
19 .text:00000018                lui     $a0, ($LC0 >> 16)  # "Hello, world!"
20 ; Springe zu puts(), speichere Ruecksprungadresse im link register:
21 .text:0000001C                jalr    $t9
22 .text:00000020                la      $a0, ($LC0 & 0xFFFF)  # "Hello, world!"
23 ; RA wiederherstellen:
24 .text:00000024                lw      $ra, 0x20+var_4($sp)
25 ; 0 von $zero zu $v0 kopieren:
26 .text:00000028                move    $v0, $zero
27 ; Ruecksrpung zu RA:
28 .text:0000002C                jr      $ra
29 ; Funktionsepilog:
30 .text:00000030                addiu   $sp, 0x20
```

Die Anweisung in Zeile 15 speichert den GP-Wert auf dem lokalen Stack. Diese Anweisung fällt seltsamerweise beim GCC, was vielleicht auf einen Fehler des Compilers hinweist. [44]. Der GP-Wert muss auch gespeichert werden weil jede Funktion ihren eigenen 64KiB-Datenbereich nutzen kann. Das Register mit der Adresse von `puts()` wird $T9, da Register mit Präfix T- temporäre Register sind deren Inhalte nicht erhalten werden müssen.

Nicht optimierender GCC

Nicht optimierender GCC ist ausführlicher.

Listing 1.32: Nicht optimierender GCC 4.4.5 (Assemblercode)

```
$LC0:
    .ascii  "Hello, world!\012\000"
main:
; Funktionsprolog.
; Sichere RA ($31) und FP auf Stack:
    addiu   $sp,$sp,-32
    sw  $31,28($sp)
    sw  $fp,24($sp)
; Setze FP (stack frame pointer):
    move    $fp,$sp
; Setze GP:
    lui $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
; Lade Adresse der Zeichenkette:
    lui $2,%hi($LC0)
    addiu   $4,$2,%lo($LC0)
; Lade Adresse von puts() mit GP:
    lw  $2,%call16(puts)($28)
    nop
; puts() aufrufen:
    move    $25,$2
```

[44] Anscheinend sind Funktionen die Listings erzeugen nicht so kritisch für GCC-Nutzer, so dass vielleicht noch unbehobene Fehler existieren.

```
    jalr    $25
    nop  ; branch delay slot

; GP vom lokalen Stack wiederherstellen:
    lw  $28,16($fp)
; Setze Register $2 ($V0) zu Null:
    move    $2,$0
; Funktionsepilog.
; SP wiederherstellen:
    move    $sp,$fp
; RA wiederherstellen:
    lw  $31,28($sp)
; FP wiederherstellen:
    lw  $fp,24($sp)
    addiu   $sp,$sp,32
; Springe zu RA:
    j   $31
    nop  ; branch delay slot
```

Es ist zu sehen, dass das FP-Register als Zeiger zum Stack Frame genutzt wird. Außerdem sind im Listing drei NOP-Anweisungen. Die zweite und dritte welche der Sprunganweisung folgt. Möglicherweise fügt der GCC-Compiler immer NOP-Anweisungen nach einer Sprung hinzu (wegen der *branch delay slots*) und entfernt diese wenn die Optimierung eingeschaltet ist. In diesem Fall bleiben sie also bestehen.

Nachfolgend das IDA-Listing:

Listing 1.33: Nicht optimierender GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; Funktionsprolog.
; Speichere RA und FP auf dem Stack:
.text:00000000                 addiu   $sp, -0x20
.text:00000004                 sw      $ra, 0x20+var_4($sp)
.text:00000008                 sw      $fp, 0x20+var_8($sp)
; Setze den FP (stack frame pointer):
.text:0000000C                 move    $fp, $sp
; Setze GP:
.text:00000010                 la      $gp, __gnu_local_gp
.text:00000018                 sw      $gp, 0x20+var_10($sp)
; Lade die Adresse der Zeichenkette:
.text:0000001C                 lui     $v0, (aHelloWorld >> 16)  # "Hello,
    world!"
.text:00000020                 addiu   $a0, $v0, (aHelloWorld & 0xFFFF)  #
    "Hello, world!"
; Lade die Adresse von puts() mit GP:
.text:00000024                 lw      $v0, (puts & 0xFFFF)($gp)
.text:00000028                 or      $at, $zero ; NOP
; Aufruf von puts():
.text:0000002C                 move    $t9, $v0
```

```
25 .text:00000030                jalr    $t9
26 .text:00000034                or      $at, $zero ; NOP
27 ; GP vom lokalen Stack wieder herstellen:
28 .text:00000038                lw      $gp, 0x20+var_10($fp)
29 ; Setze das Register $2 ($V0) zu 0:
30 .text:0000003C                move    $v0, $zero
31 ; Funktionsepilog.
32 ; SP wiederherstellen:
33 .text:00000040                move    $sp, $fp
34 ; RA wiederherstellen:
35 .text:00000044                lw      $ra, 0x20+var_4($sp)
36 ; FP wiederherstellen:
37 .text:00000048                lw      $fp, 0x20+var_8($sp)
38 .text:0000004C                addiu   $sp, 0x20
39 ; Zu RA springen:
40 .text:00000050                jr      $ra
41 .text:00000054                or      $at, $zero ; NOP
```

Interessanterweise kennt IDA das Anweisungspaar `LUI`/`ADDIU` und fasst diese zu einer einzigen Pseudoanweisung `LA` („Load Address“) zusammen (Zeile 15). Es ist auch zu sehen, dass diese Pseudoanweisung eine Größe von 8 Byte hat! Dies ist eine Pseudoanweisung (oder *Makro*) weil es sich hier nicht um eine echte MIPS-Anweisung handelt, sondern eher um einen handlichen Namen für ein Anweisungspaar.

Eine weitere Sache ist, dass IDA keine NOP-Anweisung kennt. Also ist in den Zeilen 22, 26 und 41 `OR $AT, $ZERO`. Im Wesentlichen führt diese Anweisung eine ODER-Operation auf die Inhalte des $AT-Register aus, welche 0 ist. Dies entspricht natürlich einer Idle-Anweisung. MIPS hat wie viele andere ISA keine separate NOP-Anweisung.

Aufgabe des Stack Frames in diesem Beispiel

Die Adresse dieser Zeichenkette ist in einem Register übergeben. Warum wird dennoch der lokale Stack vorbereitet? Der Grund dafür liegt in der Tatsache, dass die Werte der Register RA und GP wegen des Aufrugs von `printf()` irgendwo gesichert werden müssen und hier eben der lokale Stack dafür genutzt wird. Wenn dies eine Blatt-Funktion wäre, bestünde die Möglichkeit den Funktionsepilog und -prolog wegzulassen, wie hier: 1.4.3 on page 11.

Optimierender GCC: in GDB laden

Listing 1.34: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw
```

```
Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

1.5.5 Fazit

Der Hauptunterschied zwischen x86/ARM and x64/ARM64-Code ist das der Zeiger auf den String 64 Bit lang ist. Moderne CPUs haben eine 64-Bit-Architektur um Speicherkosten zu reduzieren und den höheren Bedarf aktueller Anwendungen erfüllen zu können. Es ist möglich sehr viel mehr Speicher in dem Computer zu verwenden als 32-Bit-Zeiger adressieren können. Aus diesem Grund sind alle Zeiger 64 Bit lang.

1.5.6 Übungen

- http://challenges.re/48
- http://challenges.re/49

1.6 Funktionsprolog und Funktionsepilog

Ein Funktionsprolog besteht aus einer Abfolge von Instruktionen zu Beginn einer Funktion. Dieser Prolog sieht häufig ähnlich aus wie das folgende Codefragment:

```
    push    ebp
    mov     ebp, esp
```

```
    sub     esp, X
```

Was diese Instruktionen tun: speichern des Wertes im EBP Register auf dem Stack, setzen des Wertes von EBP auf den Wert in ESP, zuweisen von Speicherplatz für lokale Variablen auf dem Stack.

Der Wert in EBP verändert sich während der Funktionsausführung nicht und wird für den Zugriff auf lokale Variablen und Funktionsargumente verwendet. Zum gleichen Zweck kann auch ESP verwendet werden, aber da sich der Wert von ESP während der Funktionsausführung verändert, ist diese Vorgehensweise nicht allzu gebräuchlich.

Der Funktionsepilog gibt den zugewiesenen Speicher auf dem Stack wieder frei, setzt das EBP Register auf den ursprünglichen Wert zurück und gibt den control flow an den caller zurück:

```
    mov    esp, ebp
    pop    ebp
    ret    0
```

Für gewöhnlich werden Funktionsprolog und -epilog von Disassemblern erkannt und zur Markierung und Abgrenzung von Funktionen verwendet.

1.6.1 Rekursion

Funktionsprolog und -epilog können sich negativ auf die Performanz bei Rekursion auswirken.

Mehr zum Thema Rekursion im folgenden Buch: **??** on page ??.

1.7 Stack

Der Stack ist eine der fundamentalen Datenstrukturen in der Informatik. [45]. AKA[46] **LIFO!**[47].

Technisch betrachtet ist es ein Stapelspeicher innerhalb des Prozessspeichers der zusammen mit den ESP (x86), RSP (x64) oder dem **SP!** (ARM) Register als ein Zeiger in diesem Speicherblock fungiert.

Die häufigsten Stack-Zugriffsinstruktionen sind die PUSH- und POP-Instruktionen (in beidem x86 und ARM Thumb-Modus). PUSH subtrahiert vom ESP/RSP/**SP!** 4 Byte im 32-Bit Modus (oder 8 im 64-Bit Modus) und schreibt dann den Inhalt des Zeigers an die Adresse auf die von ESP/RSP/**SP!** gezeigt wird.

POP ist die umgekehrte Operation: Die Daten des Zeigers für die Speicherregion auf die von **SP!** gezeigt wird werden ausgelesen und die Inhalte in den Instruktionsoperanden geschreiben (oft ist das ein Register). Dann werden 4 (beziehungsweise 8) Byte zum Stapel-Zeiger addiert.

[45] wikipedia.org/wiki/Call_Stack
[46] Also Known As — auch bekannt als
[47] **LIFO!**

Nach der Stackallokation, zeigt der Stapel-Zeiger auf den Boden des Stacks. PUSH verringert den Stapel-Zeiger und POP erhöht ihn. Der Boden des Stacks ist eigentlich der Anfang der Speicherregion die für den Stack reserviert wurde. Das wirkt zunächst seltsam, aber so funktioniert es.

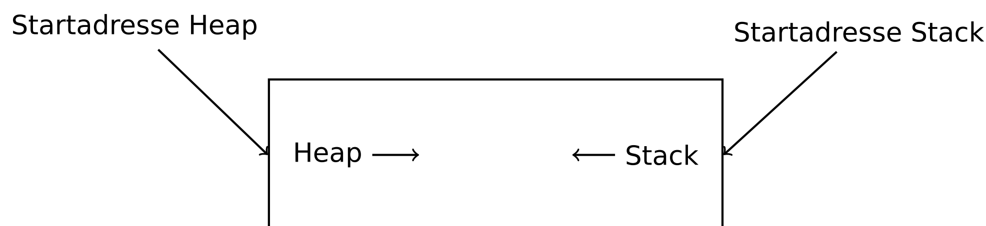
ARM unterstützt beides, aufsteigende und absteigende Stacks.

Zum Beispiel die STMFD/LDMFD und STMED[48]/LDMED[49] Instruktionen sind alle dafür gedacht mit einem absteigendem Stack zu arbeiten (wächst nach unten, fängt mit hohen Adressen an und entwickelt sich zu niedrigeren Adressen). Die STMFA[50]/LDMFA[51] und STMEA[52]/LDMEA[53] Instruktionen sind dazu gedacht mit einem aufsteigendem Stack zu arbeiten (wächst nach oben und fängt mit niedrigeren Adressen an und wächst nach oben).

1.7.1 Warum wächst der Stack nach unten?

Intuitiv, würden man annehmen das der Stack nach oben wächst z.B Richtung höherer Adressen, so wie bei jeder anderen Datenstruktur.

Der Grund das der Stack rückwärts wächst ist wohl historisch bedingt. Als Computer so groß waren das sie einen ganzen Raum beansprucht haben war es einfach Speicher in zwei Sektionen zu unterteilen, einen Teil für den Heap und einen Teil für den Stack. Sicher war zu dieser Zeit nicht bekannt wie groß der Heap und der Stack wachsen würden, während der Programm Laufzeit, also war die Lösung die einfachste mögliche.



In [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)][54]können wir folgendes lesen:

> Der user-core eines Programm Images wird in drei logische Segmente unterteilt. Das Programm-Text Segment beginnt bei 0 im virtuellen Adress Speicher. Während der Ausführung wird das Segment als schreibgeschützt markiert und eine einzelne Kopie des Segments wird unter allen Prozessen geteilt die das Programm ausführen. An der

[48]Store Multiple Empty Descending (ARM Instruktion)
[49]Load Multiple Empty Descending (ARM Instruktion)
[50]Store Multiple Full Ascending (ARM Instruktion)
[51]Load Multiple Full Ascending (ARM Instruktion)
[52]Store Multiple Empty Ascending (ARM Instruktion)
[53]Load Multiple Empty Ascending (ARM Instruktion)
[54]Auch verfügbar als URL

> ersten 8K grenze über dem Programm Text Segment im Virtuellen Speicher, fängt der “nonshared” Bereich an, der nach Bedarf von Syscalls erweitert werden kann. Beginnend bei der höchsten Adresse im Virtuellen Speicher ist das Stack Segment, das Automatisch nach unten wächst während der Hardware Stackpointer sich ändert.

Das erinnert daran wie manche Schüler Notizen zu zwei Vorträgen in einem Notebook dokumentieren: Notizen für den ersten Vortrag werden normal notiert, und Notizen zur zum zweiten Vortrag werden ans Ende des Notizbuches geschrieben, indem man das Notizbuch umdreht. Die Notizen treffen sich irgendwann im Notizbuch aufgrund des fehlenden Freien Platzes.

1.7.2 Für was wird der Stack benutzt?

1.7.3 Rückgabe Adresse der Funktion speichern

x86

Wenn man eine Funktion mit der CALL Instruktion aufruft, wird die Adresse direkt nach der CALL Instruktion auf dem Stack gespeichert und der unbedingte jump wird ausgeführt.

Die CALL Instruktion ist äquivalent zu dem PUSH `address_after_call / JMP operand` Instruktions paar.

RET ruft die Rückkehr Adresse vom Stack ab und springt zu dieser —was äquivalent zu einem POP `tmp / JMP tmp` Instruktions paar ist.

Den Stack zum überlaufen zu bringen ist recht einfach, einfach eine endlos rekursive Funktion Aufrufen:

```
void f()
{
        f();
};
```

MSVC 2008 hat eine Erkennung für das Problem:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for ↙
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, ↙
    ↳ function will cause runtime stack overflow
```

…aber der Compiler erzeugt den Code trotzdem:

```
?f@@YAXXZ PROC                          ; f
; Line 2
```

```
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ       ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                  ; f
```

…Auch wenn wir die Compiler Optimierungen einschalten (`/Ox` Option) wird der optimierte Code nicht den Stack zum überlaufen bringen. Stattdessen wird der Code *korrekt*[55] ausgeführt:

```
?f@@YAXXZ PROC                  ; f
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                  ; f
```

GCC 4.4.1 generiert vergleichbaren Code in beiden Fällen, jedoch ohne über das Overflow Problem zu warnen.

ARM

ARM Programme benutzen den Stack um Rücksprung Adressen zu speichern, aber anders. Wie bereits erwähnt in „Hallo, Welt!“ (1.5.3 on page 24), wird der RA Wert im LR (Link Register) gespeichert. Wenn nun eine andere Funktion aufgerufen werden muss und auf das LR Register zu greift, muss der aktuelle Wert im Register irgendwo gespeichert werden.

Normal wird der Wert im Funktion Prolog gespeichert.

Oft sieht man Instruktionen wie z.B `PUSH R4-R7,LR` zusammen mit dieser Instruktion im Epilog `POP R4-R7,PC`—Somit werden Werte die in den Funktionen benötigt werden auf dem Stack gespeichert, inklusive LR.

Wenn eine Funktion nie eine andere Funktion aufruft, nennt man das in der RISC Terminologie eine *leaf Funktion*[56]. Als Konsequenz ergibt sich, das leaf Funktionen nicht das LR Register speichern (da sie es nicht modifizieren). Wenn solche Funktionen klein sind und nur eine geringe Anzahl an Registern benutzt, ist es möglich das der Stack gar nicht benutzt wird. Es ist also möglich leaf Funktionen zu benutzen ohne den Stack zurück zu greifen, die Ausführung ist hier schneller als auf älteren x86 Maschinen weil kein externer RAM für den Stack benutzt wird [57] ist. Diese Eigen-

[55] Ironie hier

[56] infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[57] Bis vor einer weile war es sehr teuer auf PDP-11 und VAX Maschinen die CALL Instruktion zu benutzen; bis zu 50% der Rechenzeit wurde allein für diese Instruktion verschwendet, man hat dabei festgestellt das eine große Anzahl an kleinen Funktionen zu haben ein Anti-Pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

schaft kann nützlich sein wenn der Speicher für den Stack noch nicht alloziert oder verfügbar ist.

Ein paar Beispiele für leaf Funktionen:

1.10.3 on page 117, 1.10.3 on page 117, 1.255 on page 368, 1.271 on page 390, 1.21.5 on page 390, 1.165 on page 243, 1.163 on page 240, 1.182 on page 263.

Funktion Argumente übergeben

Der übliche weg Argumente in x86 zu übergeben ist die „cdecl" Methode:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

Die Callee Funktionen bekommen ihre Argumente über den Stackpointer.

So werden die Argumente auf dem Stack gefunden, noch vor der Ausführung der ersten Instruktion der f() Funktion:

ESP	return address
ESP+4	Argument#1, in IDA gekennzeichnet als arg_0
ESP+8	Argument#2, in IDA gekennzeichnet als arg_4
ESP+0xC	Argument#3, in IDA gekennzeichnet als arg_8
…	…

Für mehr Informationen über andere Aufrufs Konventionen siehe Sektion: (6.1 on page 580).

Übrigens, die callee Funktion hat keine Informationen wie viele Argumente übergeben wurden. C Funktionen mit einer variablen Anzahl an Argumenten (wie z.B printf()) errechnen die zahl der Argumente anhand der Formatstring spezifizier-er (alle spezifizier-er die mit dem % beginnen).

Wenn wir etwas schreiben wie z.B:

```
printf("%d %d %d", 1234);
```

printf() wird die Zahlen 1234 und zwei zufällige Werte ausgeben, welche direkt neben 1234 auf dem Stack lagen[58].

Das ist auch der Grund warum es nicht wichtig ist wie die main() Funktion definiert ist: Als main(),
main(int argc, char *argv[]) oder main(int argc, char *argv[], char *envp[]).

Tatsächlich ruf der CRT-Code die main() Funktion um Grunde so auf:

```
push envp
push argv
push argc
```

[58] Nicht zufällig im eigentlichen Sinne sondern eher unvorhersehbar: 1.7.5 on page 51

```
call main
...
```

Wenn man `main()` als `main()` Funktion ohne Argumente definiert, dann liegen sie trotzdem auf dem Stack auch wenn sie nicht benutzt werden. Wenn man `main()` als `main(int argc, char *argv[])`, definiert kann man auf die ersten beiden Argumente der Funktion zugreifen, das dritte bleibt aber weiterhin “Unsichtbar” für andere Funktionen. Es ist aber auch u.a möglich die Main Funktion als `main(int argc)` schreiben und sie wird noch immer funktionieren.

Alternative Wege Argumente zu übergeben

Es sollte bemerkt werden das nichts einen Programmierer dazu zwingt Argumente über den Stack zu übergeben. Das ist keine generelle Anforderung. Jemand könnte auch einfach eine andere Methode implementieren ohne den Stack überhaupt zu benutzen.

Ein ziemlich beliebter Weg Argumente zu übergeben unter Assembler Neulingen ist über globale Variablen wie z.B:

Listing 1.35: Assembly code

```
        ...

        mov     X, 123
        mov     Y, 456
        call    do_something

        ...

X       dd      ?
Y       dd      ?

do_something proc near
        ; take X
        ; take Y
        ; do something
        retn
do_something endp
```

Aber diese Methode hat Nachteile: Die *do_something()* Funktion kann sich selbst nicht rekursiv aufrufen (aber auch keine andere Funktion), weil sie ihre eigenen Argumente löschen muss. Die gleiche Geschichte mit lokalen Variablen: Wenn die Werte in globalen Variablen gespeichert sind, kann die Funktion sich nicht selbst aufrufen. Und das bedeutet wiederum das die Funktion nicht thread-Safe ist. [59]. Eine Methode solche Informationen auf dem Stack zu speichern macht die Dinge einfacher— Der Stack kann so viele Funktion Arguemente und/oder Werte speichern, so viel Speicher wie der Computer hat.

[59] Korrekt implementiert, hat jeder Thread seinen eigenen Stack und seine eigenen Argumente/Variablen

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] nennt sogar noch verrückter Methoden die speziell auf IBM System/360 benutzt werden.

Auf MS-DOS gab es einen Weg Funktion Argumente über Register zu übergeben, zum Beispiel dies ist ein Stück Code einer veralteten 16-Bit MS-DOS “Hallo, Welt!” Funktion:

```
mov  dx, msg      ; Adresse der Naricht
mov  ah, 9        ; 9 bedeutet ``print string''
int  21h          ; DOS "syscall"

mov  ah, 4ch      ; ``Terminiere Programm'' Funktion
int  21h          ; DOS "syscall"

msg  db 'Hello, World!\$'
```

Diese Methode ist der 6.1.3 on page 582 Methode sehr ähnlich. Sie ähnelt aber auch der Methode wie man auf Linux (6.3.1 on page 599) und Windows syscalls ausführt.

Wenn eine MS-DOS Funktion einen Bool’schen Wert zurück gibt (z.B., Single Bit bedeutet ein Fehler ist aufgetreten), wird dafür das CF Flag benutzt.

Zum Beispiel:

```
mov ah, 3ch       ; create file
lea dx, filename
mov cl, 1
int 21h
jc  error
mov file_handle, ax
...
error:
...
```

Im Falle eines Fehlers, wird das CF Flag gesetzt. Anderenfalls wird ein handle für die neu erstellte Datei über AX zurück gegeben.

Diese Methode wird heute immer noch von Assembler Programmierern benutzt. Im Windows Reseearch Kernel source Code (der sehr ähnlich zum Windows 2003 Kernel ist) können wir folgenden Code finden (file *base/ntos/ke/i386/cpu.asm*):

```
        public  Get386Stepping
Get386Stepping  proc

        call    MultiplyTest            ; Muliplikations Test durchführen
        jnc     short G3s00             ; wenn nc, ist muttest ok
        mov     ax, 0
        ret
G3s00:
        call    Check386B0              ; Prüfe das B0 stepping
        jnc     short G3s05             ; wenn nc, ist es B1/later
        mov     ax, 100h                ; It is B0/earlier stepping
        ret
```

```
G3s05:
        call    Check386D1              ; Prüfe das D1 stepping
        jc      short G3s10             ; wenn c, iust es NICHT NOT D1
        mov     ax, 301h                ; Es ist das D1/later stepping
        ret

G3s10:
        mov     ax, 101h                ; annahme das es das it is B1
    stepping ist
        ret

        ...

MultiplyTest    proc

        xor     cx,cx                   ; 64K durchläufe ist eine nette runde
    Nummer
mlt00:  push    cx
        call    Multiply                ; Funktioniert dis multiplikation auf
    diesem Chip?
        pop     cx
        jc      short mltx              ; wenn c c, Nein, exit
        loop    mlt00                   ; Wenn nc, Ja, weitere iteration für
    nächsten versuch
        clc
mltx:
        ret

MultiplyTest    endp
```

Local variable storage

Eine Funktion kann platz für lokale Variablen allokieren in dem sie einfach den Stapel-Zeiger verkleinert in richtung der niedrigsten Adresse des Stacks verschiebt.

Dieser Weg ist ziemlich schnell, egal wie viele Variablen deffiniert werden. Es ist aber keine Anforderung lokale Variablen auf dem Stack zu speichern. Man kann lokale Variablen speicher wo immer man will, aber traditionell speichert man sie auf dem Stack.

x86: alloca() Funktion

Es macht Sinn einen Blick auf die `alloca()` Funktion zu werfen [60] gefunden werden. Diese Funktion arbeitet wie `malloc()`, nur das sie Speicher direkt auf dem Stack bereit stellt.

Der allozierte Speicher Chunk muss nicht wieder mit `free()` freigegeben werden, weil der Funktions Epilog (1.6 on page 39) das ESP Register wieder in seinen ursprünglichen Zustand versetzt und der allozierte Speicher wird einfach *verworfen*.

[60] In MSVC, kann die Funktions Implementierung in `alloca16.asm` und `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

Es macht Sinn sich anzuschauen wie `alloca()` implementiert ist. Mit einfachen Begriffen erklärt, diese Funktion verschiebt ESP in Richtung des Stack ende mit der Anzahl der Bytes die alloziert werden müssen und setzt ESP als einen Zeiger auf den *allozierten* block.

Beispiel:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Die `_snprintf()` Funktion arbeitetet genau wie `printf()`, nur statt die Ergebnisse nach stdout aus zu geben (bsp. auf dem Terminal oder Konsole), schreibt sie in den `buf` buffer. Die Funktion `puts()` kopiert den Inhalt aus `buf` nach stdout. Sicher könnte man die beiden Funktions Aufrufe könnten durch einen `printf()` Aufruf ersetzt werden, aber wir sollten einen genaueren Blick auf die Benutzung kleiner Buffer anschauen.

MSVC

Compilierung mit MSVC 2010:

Listing 1.36: MSVC 2010

```
...

    mov     eax, 600  ; 00000258H
    call    __alloca_probe_16
    mov     esi, esp

    push    3
    push    2
    push    1
    push    OFFSET $SG2672
    push    600       ; 00000258H
    push    esi
    call    __snprintf
```

```
    push    esi
    call    _puts
    add     esp, 28

...
```

Das einzige
TTalloca() Argument wird über EAX übergeben (anstatt es erst auf den Stack zu pushen) [61].

GCC + Intel Syntax

GCC 4.4.1 macht das selbe, aber ohne externe Funktions aufrufe.

Listing 1.37: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                    ; Pointer an 16-byte grenze
    anpassen
        mov     DWORD PTR [esp], ebx        ; s
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600      ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx        ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + AT&T Syntax

Nun der gleiche Code, aber in AT&T Syntax:

[61] Das liegt daran, das alloca() Verhalten Compiler intrinsisch bestimmt (10.4 on page 666) im Gegensatz zu einer normalen Funktion. Einer der Gründe dafür das man braucht eine separate Funktion braucht, statt ein paar Code Instruktionen im Code, ist weil die **MSCV!**[62] alloca() Implementierung ebenfalls Code hat welcher aus dem gerade allozierten Speicher gelesen wird. Damit in Folge das **Betriebssystem!**[63] physikalischen Speicher in dieser VM[64] Region zu allozieren. Nach dem `alloca()` Aufruf, zeigt ESP auf den Block von 600 Bytes der nun als Speicher für das `buf` Array dienen kann.

Listing 1.38: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

Der Code ist der gleiche wie im vorherigen listig.

Übrigens, `movl $3, 20(%esp)` in AT&T Syntax wird zu `mov DWORD PTR [esp+20], 3` in Intel-syntax. In der AT&T Syntax, sehen Register+Offset Formatierungen einer Adresse so aus: `offset(%register)`.

(Windows) SEH

Automatisches deallokieren der Daten auf dem Stack

Vielleicht ist der Grund warum man lokale Variablen und SEH Einträge auf dem Stack speichert, weil sie beim verlassen der Funktion automatisch aufgeräumt werden. Man braucht dabei nur eine Instruktion um die Position des Stackpointers zu korrigieren (oftmals ist es die ADD Instruktion). Funktions Argumente, könnte man sagen werden auch am Ende der Funktion deallokiert. Im Kontrast dazu, alles was auf dem *heap* gespeichert wird muss explizit deallokiert werden.

1.7.4 Ein typisches Stack Layout

Ein typisches Stacklayout auf einer 32-Bit Umgebung sieht am Anfang der ausführung einer Funktion, noch bevor der ausführung der ersten Instruktion wie folgt aus:

…	…
ESP-0xC	lokale Variable#2, in IDA gekennzeichnet als var_8
ESP-8	lokale Variable#1, in IDA gekennzeichnet als var_4
ESP-4	abgespeicherter Wert vonEBP
ESP	Rücksprungadresse
ESP+4	Argument#1, in IDA gekennzeichnet als arg_0
ESP+8	Argument#2, in IDA gekennzeichnet als arg_4
ESP+0xC	Argument#3, in IDA gekennzeichnet als arg_8
…	…

1.7.5 Rauschen auf dem Stack

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

Stephen Wolfram, A New Kind of Science.

Oft wird in diesem Buch von „rauschen“ oder „garbage“ Werten im Bezug auf den Stack gesprochen. Woher kommen diese Werte? Das sind Überbleibsel der Ausführung von anderen Funktionen. Zum Beispiel:

```
#include <stdio.h>

void f1()
{
        int a=1, b=2, c=3;
};

void f2()
{
        int a, b, c;
        printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
        f1();
        f2();
};
```

Compilieren …

Listing 1.39: Nicht optimierender MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f1     PROC
        push    ebp
        mov     ebp, esp
```

```
        sub     esp, 12
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     esp, ebp
        pop     ebp
        ret     0
_f1     ENDP

_c$ = -12       ; size = 4
_b$ = -8        ; size = 4
_a$ = -4        ; size = 4
_f2     PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 12
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        push    OFFSET $SG2752 ; '%d, %d, %d'
        call    DWORD PTR __imp__printf
        add     esp, 16
        mov     esp, ebp
        pop     ebp
        ret     0
_f2     ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        call    _f1
        call    _f2
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Hier wird sich der Compiler ein bisschen beschweren…

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for ⤦
    ↳ 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' ⤦
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' ⤦
    ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' ⤦
    ↳ used
```

```
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:st.exe
st.obj
```

Aber wenn wir das compilierte Programm laufen lassen…

```
c:\Polygon\c>st
1, 2, 3
```

sieh an! Wir haben keine Variablen gesetzt in `f2()`. Das sind „Geister“ Werte, welche noch immer auf dem Stack rumliegen.

Lasst uns das Beispiel in OllyDbg laden:

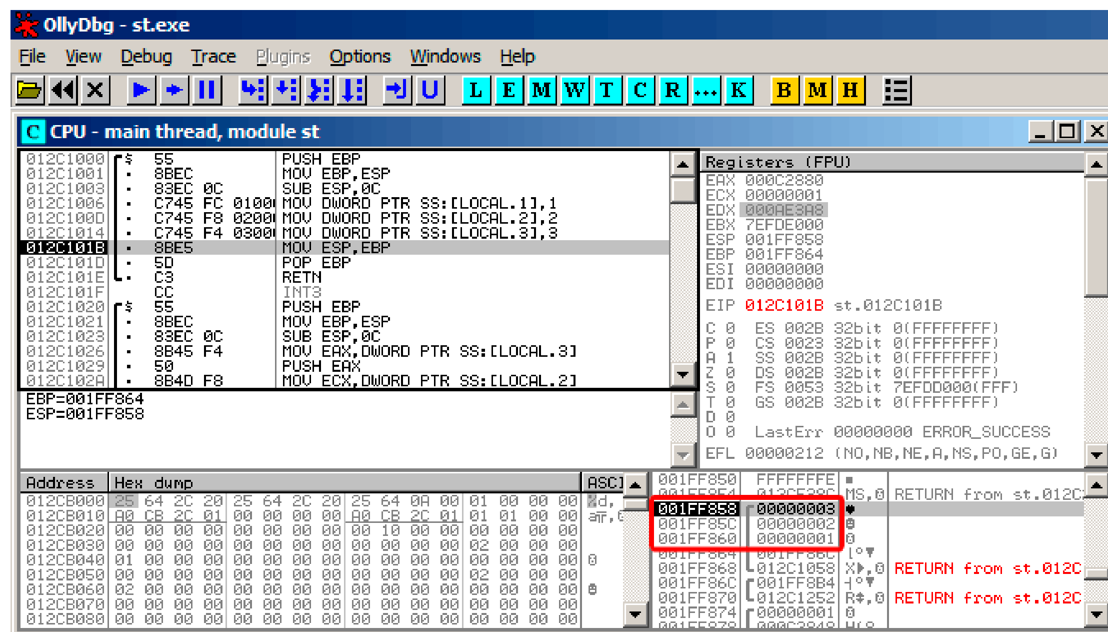


Abbildung 1.6: OllyDbg: `f1()`

Wenn `f1()` den Variablen a, b und c ihre Werte zuordnet, wird ihre Adresse bei `0x1FF860` gespeichert und so weiter.

Und wenn f2() ausgeführt wird:

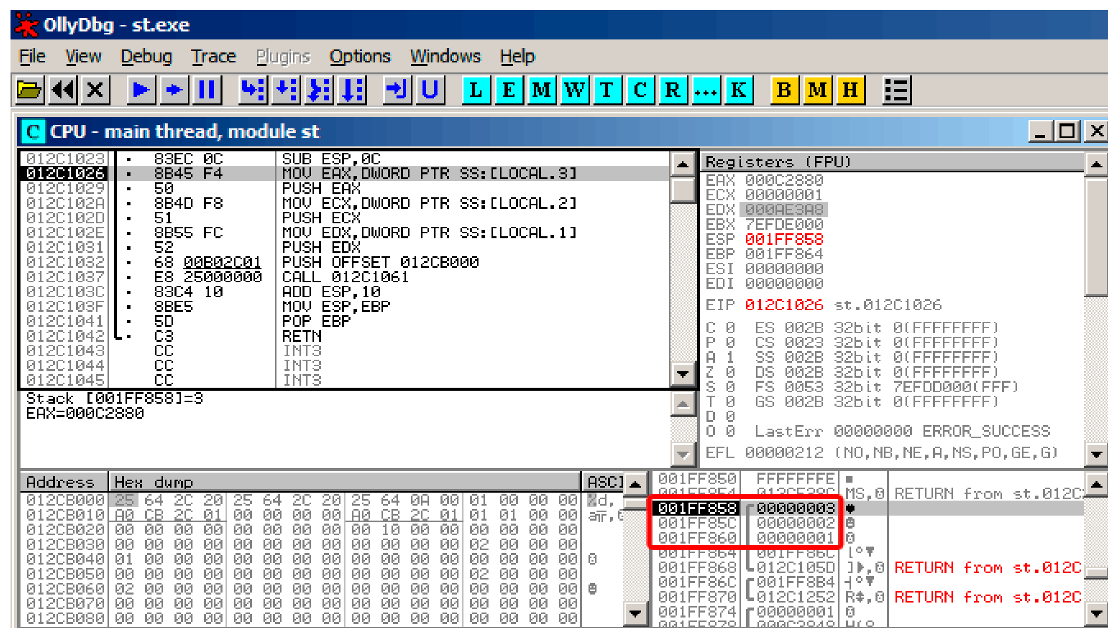


Abbildung 1.7: OllyDbg: f2()

... liegen a, b und c von f2() an den gleichen Adressen! Nichts hat bis jetzt Ihre Werte überschrieben und sie sind bisher unberührt geblieben. Also, damit diese seltsame Situation eintritt, müssen mehrere Funktionen nacheinander aufgerufen werden und **SP!** muss gleich sein für jede Funktions Instruktion (z.B. die Funktionen haben die gleiche Anzahl an Argumenten). Dann werden die lokalen Variablen an den gleichen Positionen im Stack liegen. Zusammen fassend kann man sagen, alle Werte auf dem Stack (und Speicherzellen im allgemeinen) beinhalten Werte von vorhergehenden Funktions aufrufen. Diese Werte sind nicht zufällig im klassischem Sinn, eher unvorhersehbar. Es wäre wahrscheinlich möglich Teile des Stacks auf zu räumen vor jedem Funktions Aufruf, aber das wäre zu viel zusätzliche (und unnötige) Arbeit.

MSVC 2013

Das Beispiel wurde compiliert mit dem MSVC 2010 Compiler. Allerdings haben die Leser dieses Buch auch schon geschafft das Beispiel mit MSVC 2013 zu compilieren, sie haben es geschafft es zum laufen zu bringen und alle drei Nummern zu reversen.

```
c:\Polygon\c>st
3, 2, 1
```

Warum? Ich habe das Beispiel auch mit MSCV 2013 compiliert und habe folgendes beobachtet:

Listing 1.40: MSVC 2013

```
_a$ = -12        ; size = 4
_b$ = -8         ; size = 4
_c$ = -4         ; size = 4
_f2     PROC

...

_f2     ENDP

_c$ = -12        ; size = 4
_b$ = -8         ; size = 4
_a$ = -4         ; size = 4
_f1     PROC

...

_f1     ENDP
```

Im Gegensatz zu MSVC 2010, alloziert MSCV 2013 die Variablen in der Funktion `f2()` in umgekehrter Reihenfolge.Was auch vollkommen Korrekt ist, weil es im C/C++ Standard keine Vorschriften gibt, in welcher Reihenfolge Variablen auf dem Stack alloziert werden müssen. Der Grund für den Unterschied liegt daran das MSCV 2010 eine Methode genutzt hat um die allozierung durch zu führen und in MSCV 2013 wurde scheinbar eine anpassung im Compiler inneren gemacht, so das sich MSCV 2013 leicht anders verhält.

1.7.6 Übungen

- http://challenges.re/51
- http://challenges.re/52

1.8 printf() mit mehreren Argumenten

An dieser Stelle wird das *Hallo, Welt!* (1.5 on page 11)-Beispiel ein wenig erweitert, indem `printf()` in der `main()`-Funktion durch folgendes ersetzt wird:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

1.8.1 x86

x86: 3 Argumente

MSVC

Beim Kompilieren mit MSVC 2010 Express wird folgende Ausgabe erzeugt:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
        add     esp, 16                                 ; 00000010H
```

Fast die gleiche Ausgabe, allerdings sind jetzt die `printf()`-Argumente in umgekehrter Reihenfolge auf dem Stack hinterlegt. Das erste Argument kommt zuerst.

Die Variablen vom Typ *int* sind in einer 32-Bit-Umgebung 32 Bit breit, was 4 Byte entspricht.

In diesem Fall sind 4 Argumente vorhanden. $4 * 4 = 16$ —diese benötigen exakt 16 Byte auf dem Stack einen 32-Bit-Zeiger auf eine Zeichenkette und 3 Zahlen des Typs *int*.

Wenn der Stapel-Zeiger (ESP-Register) nach einem Funktionsaufruf durch die Anweisung `ADD ESP, X` wiederhergestellt wird, wird die Anzahl Argumente oft einfach durch die Division von X durch 4 abgeleitet.

Natürlich ist dies eine Eigenheit der *cdecl*-Aufrufkonvention und nur für 32-Bit-Umgebungen gültig (siehe auch Abschnitt über Aufrufkonventionen (6.1 on page 580)).

In bestimmten Fällen in denen mehrere Funktionen direkt hintereinander beendet werden, kann der Compiler mehrere „ADD ESP, X“-Anweisungen in einer zusammenfassen:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Hier ist ein Beispiel aus einer realen Applikation:

Listing 1.41: x86

```
.text:100113E7    push    3
.text:100113E9    call    sub_100018B0 ; erwartet ein Argument (3)
.text:100113EE    call    sub_100019D0 ; erwartet kein Argument
.text:100113F3    call    sub_10006A90 ; erwartet kein Argument
.text:100113F8    push    1
.text:100113FA    call    sub_100018B0 ; erwartet ein Argument (1)
.text:100113FF    add     esp, 8       ; Addiert zwei Argumente auf einmal
    auf den Stack
```

MSVC und OllyDbg

Sehen wir uns das Beispiel in OllyDbg an, der einer der populärsten User-Land-Win32-Debugger ist. Wenn das Beispiel in MSVC 2012 mit der Option /MD kompiliert wird, wird gegen MSVCR*.DLL gelinkt. Somit kann die importierte Funktion klar im Debugger gesehen werden.

Anschließend wird die ausführbare Datei in OllyDbg geladen. Der erste Breakpoint ist in ntdll.dll, F9 startet die Ausführung. Der zweite Breakpoint ist im CRT-Code.

Jetzt muss die main()-Funktion gefunden werden, die sich ganz oben im Code befindet (MSVC allokiert die main()-Funktion ganz zu Beginn der Code-Sektion):

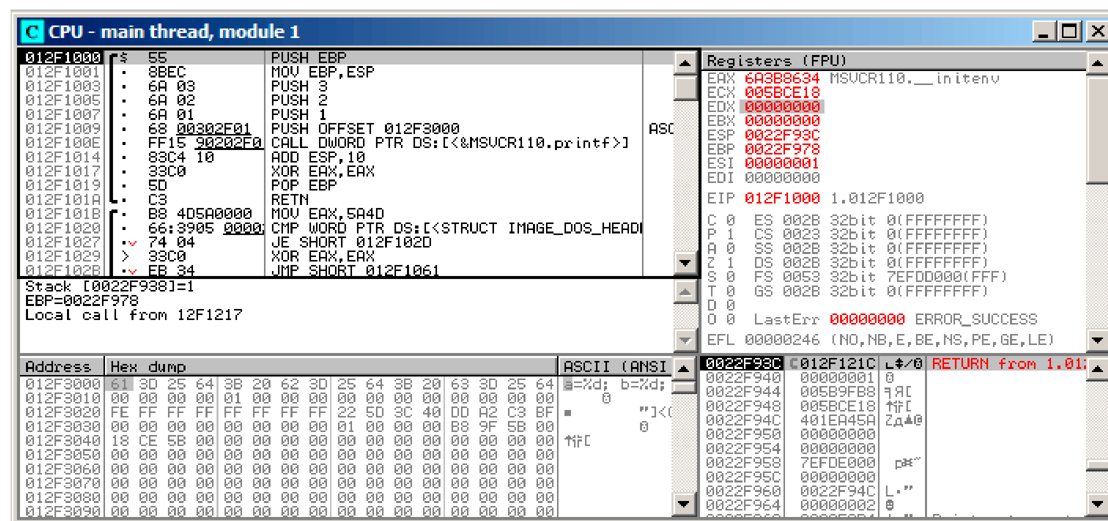


Abbildung 1.8: OllyDbg: Der Start der main()-Funktion

Um den CRT-Code zu überspringen (der hier nicht von Interesse ist) müssen folgende Schritte ausgeführt werden: Klicken auf die PUSH EBP-Anweisung, Drücken von F2 (Breakpoint setzen) und Drücken von F9 (Starten

Sechsmaliges Drücken von F8 (step over) überspringt sechs Anweisungen:

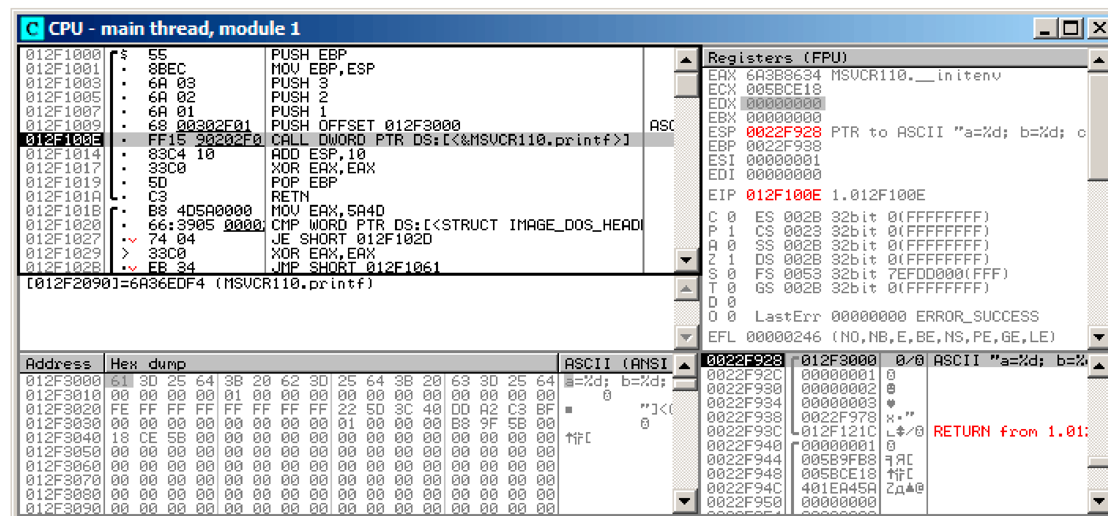


Abbildung 1.9: OllyDbg: vor der Ausführung von`printf()`

Jetzt zeigt der **PC!** auf die `CALL printf`-Anweisung. OllyDbg hebt, wie andere Debugger auch, den Wert des Registers hervor, welches sich geändert hat. Bei jedem Drücken von F8 ändert sich also EIP und der Wert wird in rot angezeigt. ESP ändert sich ebenfalls, weil die Werte der Argumente auf dem Stack gesichert werden.

Wo befinden sich die Werte auf dem Stack? Ein Blick auf das untere, rechte Fenster des Debuggers liefert die Antwort:

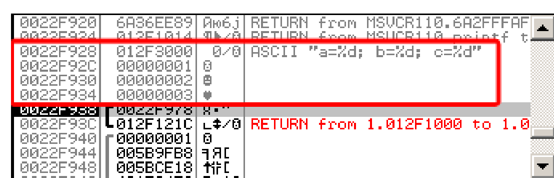


Abbildung 1.10: OllyDbg: Stack nachdem der Wert des Arguments gesichert wurde (Das rote Rechteck wurde vom Autor hinzugefügt)

Es sind hier drei Spalten sichtbar: Die Adresse im Stack, der Wert im Stack und einige weitere OllyDbg-Kommentare. OllyDbg versteht `printf()`-ähnliche Zeichenketten und zeigt sie zusammen mit den drei *angehangenen* Werten an.

Es ist möglich einen Rechtsklick auf den Formatstring und anschließend auf „Follow in dump" zu klicken. Der Formatstring wird in dem linken-unterem Debugger erscheinen, welches immer einen Teil des Speichers zeigt. Diese Speicherwerte können editiert werden. Es ist möglich den Formatstring zu verändern, was die Ausgabe dieses Beispiels ebenfalls verändern würde. In diesem Fall ist das nicht sehr nützlich, aber

es könnte eine gute Übung sein um ein Gefühl dafür zu bekommen wie hier alles funktioniert.

Drücken von F8 (step over).

Es erscheint die folgende Ausgabe auf der Konsole:

```
a=1; b=2; c=3
```

Nachfolgend die Änderungen in den Registern und der Status des Stacks:

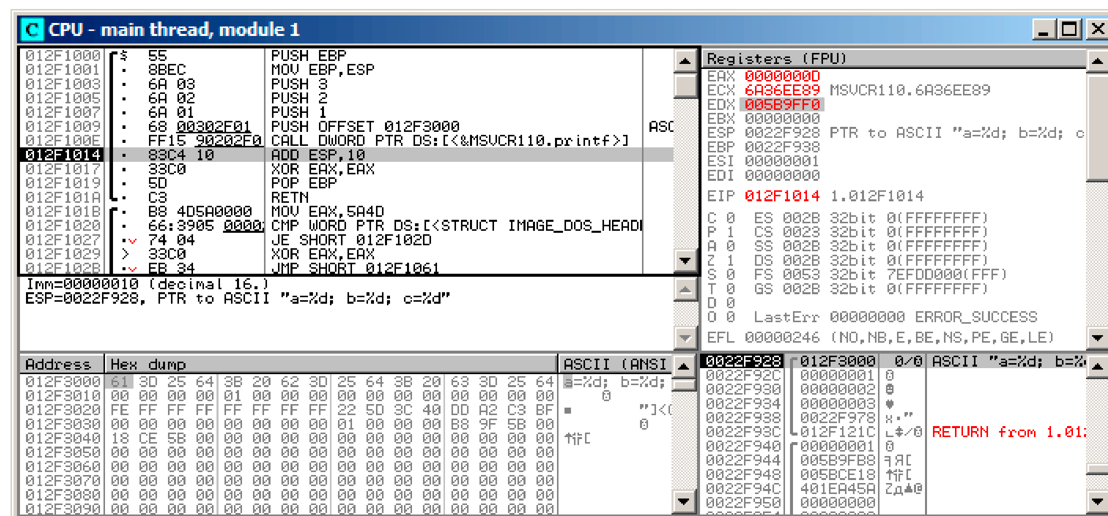


Abbildung 1.11: OllyDbg nach der Ausfühung von `printf()`

Das Register EAX enthält nun `0xD` (13). Dies ist auch richtig, weil `printf()` die Anzahl der ausgegeben Zeichen zurück gibt. Der Wert von EIP hat sich verändert: er enthält nun die Adresse der Anweisung, die nach `CALL printf` kommt. Die Werte von ECX und EDX haben sich ebenfalls geändert. Offensichtlich nutzt die `printf()`-Funktion diese für interne Zwecke.

Ein wichtiger Punkt ist, dass weder der ESP-Wert, noch der Status des Stacks sich geändert haben! Es ist klar erkennbar, dass der Formatstring und die drei entsprechenden Werte immer noch da sind. Dies ist das Verhalten der *cdecl*-Aufrufkonvention: callee setzt ESP nicht auf den vorherigen Wert zurück. Dies ist die Aufgabe der caller.

Durch erneutes Drücken von F8 wird die Anweisung ADD ESP, 10 ausgeführt:

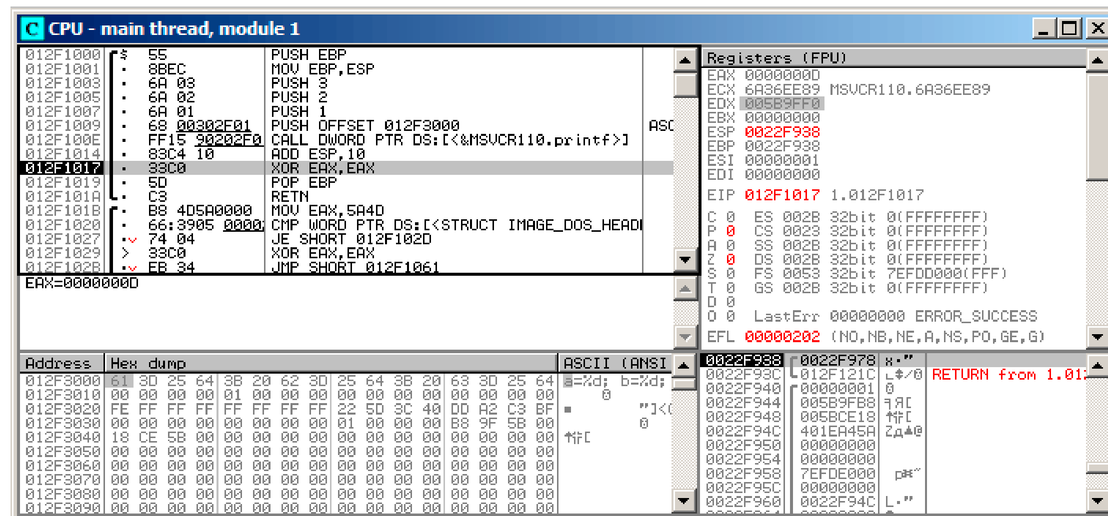


Abbildung 1.12: OllyDbg: Nach der Ausführung von ADD ESP, 10

ESP hat sich geändert, aber die Werte sind immer noch auf dem Stack! Das liegt natürlich daran, dass es keine Notwendigkeit gibt diese Werte auf Null zu setzen oder Ähnliches. Alle oberhalb des Stack-Pointers (**SP!**) ist *Rauschen* oder *Müll* und hat keine Bedeutung. Es wäre zeitintensiv und unnötig die ungenutzten Stack-Einträge zurück zu setzen.

GCC

Nun wird das gleiche Programm unter Linux mit GCC 4.4.1 kompiliert und in IDA untersucht:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
```

```
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Es ist erkennbar, dass der Unterschied zwischen MSVC- und GCC-Code lediglich die Art ist, auf der die Argumente auf dem Stack gespeichert werden. Hier arbeitet GCC direkt mit dem Stack ohne Benutzung von PUSH/POP.

GCC und GDB

Nachfolgend das Beispiel mit GDB[65] unter Linux.

Die Option -g weist den Compiler an, Debugging-Informationen in die ausführbare Datei einzufügen.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.42: Setzen eines Breakpoints auf printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Ausführen. Im Quellcode ist keine printf()-Funktion, also kann GDB sie nicht anzeigen.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

Ausgeben von 10 Stack-Einträgen. Die Spalte ganz links enthält die Adressen auf dem Stack.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

Das allererste Element ist RA (0x0804844a). Dies kann durch das Disassemblieren des Speichers an dieser Stelle überprüft werden:

[65] GNU Debugger

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Die beiden XCHG-Anweisungen sind Idle-Anweisungen, analog zu NOP.

Das zweite Element (0x080484f0) ist die Adresse des Formatstrings:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Die nächsten drei Elemente (1, 2, 3) sind die printf()-Argumente. Der Rest der Elemente kann lediglich „garbage“ auf dem Stack sein, oder auch Werte von anderen Funktionen, deren lokale Variablen usw. An dieser Stelle können sie ignoriert werden.

Ausführen von „finish“. Dieses Kommando führt dazu, dass GDB „alle Anweisungen bis zum Ende der Funktion ausführt“. In diesem Fall: Ausführen bis zum Ende von printf().

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ↙
    ↘ printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB zeigt was printf() in EAX zurück gibt (13). Die ist die Anzahl der ausgegebenen Zeichen, genau wie im OllyDbg-Beispiel.

Sichtbar ist auch „return 0;“ und die Information das dieser Ausdruck in der Datei 1.c in Zeile 6 ist. 1.c befindet sich in aktuellen Verzeichnis und GDB kann die Zeichenkette dort finden. Wie erkennt GDB welche C-Zeile gerade ausgeführt wird? Dies kommt von der Tatsache, dass der Compiler beim Generieren der Debugging-Informationen auch eine Tabelle mit Verweisen zwischen dem Quellcode-Zeilen und den Anweisungsadressen anlegt. GDB ist also ein Debugger auf Quellcode-Ebene.

Schauen wir uns die Register an. 13 in EAX:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Nachfolgend das Disassemblieren der aktuellen Anweisung. Der Pfeil zeigt auf die nächste Anweisung die ausgeführt wird.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB nutzt standardmäßig den AT&T-Syntax. Es ist aber möglich auf den Intel-Syntax zu wechseln:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Ausführen der nächsten Anweisung. GDB zeigt Ende-Klammern, welche den Block schließt.

```
(gdb) step
7       };
```

Das Ansehen der Register nach der `MOV EAX, 0`-Anweisung zeigt. dass das EAX-Register an dieser Stelle Null ist.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
```

```
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

1.8.2 ARM

1.8.3 Fazit

Hier ist der grobe Aufbau der Aufruffunktion:

Listing 1.43: x86

```
...
PUSH Drittes Argument
PUSH Zweites Argument
PUSH Erstes Argument
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

Listing 1.44: x64 (MSVC)

```
MOV RCX, Erstes Argument
MOV RDX, Zweites Argument
MOV R8, Drittes Argument
MOV R9, Viertes Argument
...
PUSH fünftes, sechstes Argument, usw. (falls notwendig)
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

Listing 1.45: x64 (GCC)

```
MOV RDI, Erstes Argument
MOV RSI, Zweites Argument
MOV RDX, Drittes Argument
MOV RCX, Viertes Argument
MOV R8, Fünftes Argument
MOV R9, Sechstes Argument
...
PUSH Siebtes, Achtes Argument, usw. (falls notwendig)
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

Listing 1.46: ARM

```
MOV R0, Erstes Argument
MOV R1, Zweites Argument
MOV R2, Drittes Argument
MOV R3, Viertes Argument
; Fünftes, Sechstes Argument, usw. auf den Stack (falls notwendig)
```

```
BL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

Listing 1.47: ARM64

```
MOV X0, Erstes Argument
MOV X1, Zweites Argument
MOV X2, Drittes Argument
MOV X3, Viertes Argument
MOV X4, Fünftes Argument
MOV X5, Sechstes Argument
MOV X6, Siebtes Argument
MOV X7, Achtes Argument
; Neuntes, Zehntes Argument, usw. auf den Stack (falls notwendig)
BL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

Listing 1.48: MIPS (O32 calling convention)

```
LI $4, Erstes argument ; AKA $A0
LI $5, Zweites argument ; AKA $A1
LI $6, Drittes argument ; AKA $A2
LI $7, Viertes argument ; AKA $A3
; pass Fünftes, Sechstes argument, usw. auf den Stack (falls notwendig)
LW temporäres Register, Adresse der Funktion
JALR temporäres Regist
```

1.8.4 Übrigens…

Übrigens ist der Unterschied der Art der Argumenten Übergabe in x86, x64, fastcall, ARM und MIPS eine gute Darstellung der Tatsache, dass die CPU nicht weiß wie die Argumente an die Funktion übergeben werden. Es ist auch möglich einen hypothetischen Compiler zu erstellen, der die Möglichkeit hat Argumente mittels einer speziellen Struktur, ohne den Stack an die Funktionen zu übergebe.

MIPS $A0 …$A3-Register sind aus Bequemlichkeitsgründen auf diese Weise beschriftet (O32 Aufrufkonvention). Programmierer können auch andere Register (vielleicht außer $ZERO) nutzen um Daten zu übergeben oder eine andere Aufrufkonvention zu nutzen.

Die CPU hatte jedoch keinerlei Kenntnisse über die Aufrufkonvention.

Man sieht hier auch wie Neulinge der Assemblersprache Argumente an andere Funktionen übergeben: in der Regel per Register ohne explizite Reihenfolge oder globale Variablen. Natürlich funktioniert das ebenso gut.

1.9 scanf()

1.9.1 Ein einfaches Beispiel

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

Es ist nicht ratsam `scanf()` heutzutage noch für User Interaktionen zu verwenden. Aber dennoch können wir hier die Übergabe eines Pointers an eine Variable vom Typ *int* betrachten.

Pointer

Pointer sind eines der fundamentalen Konzepte in der Informatik. Oft ist das Übergeben eines großen Array, eines Structs oder Objekts als Funktionsargument zu teuer, während die Übergabe der Adresse wesentlich billiger ist. Wenn man zum Beispiel einen Textstring auf der Konsole ausgeben möchte, ist es deutlich einfacher, nur dessen Adresse in den Kernel des BS zu übergeben.

Wenn die aufgerufene Funktion außerdem das große Array oder Struct verändern muss und das gesamte Object zurückgeben muss, ist die Situation beinahe absurd. Das einfachste ist also die Adresse eines Arrays oder Structs an die aufgerufene Funktion zu übergeben und sie dann die notwendigen Veränderungen durchführen zu lassen.

Ein Pointer ist in C/C++ nichts anderes als die Adresse einer Speicherstelle.

In x86 wird die Adresse als 32-Bit-Zahl dargestellt, d.h. sie benötigt 4 Byte, während in x86-64 eine Darstellung durch 64 Bit (d.h. 8 Byte) erfolgt. Dies ist übrigens der Grund dafür, dass einige Leute den Wechsel zu x86-64 ablehnen–alle Pointer in der x64-Architektur erfordern doppelt soviel Speicherplatz, inklusive Speicher in Cache, der ein sehr teurer Speicher ist.

Es ist möglich lediglich mit untypisierten Pointern zu arbeiten, wenn man ein wenig zusätzlichen Aufwand betreibt; z.B. in der Standard-C-Funktion `memcpy()`, die einen Datenblock von einer Speicherstelle zu einer anderen kopiert, werden zwei Pointer vom Typ `void*` als Argumente verwendet, da es nicht vorhersagbar ist, welchen Datentyp die Funktion kopieren soll. Datentypen sind hier nicht wichtig, entscheidend ist hier nur die Größe des Speicherblocks.

Pointer werden außerdem häufig verwendet, wenn eine Funktion mehr als einen Wert zurückgeben muss. (Darauf kommen wir später in (1.12 on page 125) zurück.)

Die Funktion *scanf()* ist solch ein Fall: Neben der Tatsache, dass die Funktion angeben muss wie viele Werte erfolgreich gelesen wurden, muss sie auch alle diese Werte

zurückliefern.

In C/C++ wird der Pointertyp nur für Typüberprüfungen zur Compilezeit benötigt.

Intern steckt im kompilierten Code keinerlei Information über die Typen der enthaltenen Pointer.

x86

MSVC

Den folgenden Code erhalten wie nach dem Kompilieren mit MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Compiler Flags der Funktion: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
    add    esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push   ecx
    push   OFFSET $SG3833 ; 'You entered %d...'
    call   _printf
    add    esp, 8

    ; return 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

x ist eine lokale Variable.

Gemäß dem C/C++-Standard darf diese nur innerhalb dieser Funktion sichtbar sein und nicht aus einem anderen, äußeren Scope. Traditionell werden lokale Variablen

auf dem Stack gespeichert. Es gibt möglicherweise andere Wege sie anzulegen, aber in x86 geschieht es auf diese Weise.

Das Ziel des Befehls direkt nach dem Funktionsprolog, PUSH ECX), ist es nicht, den Status von ECX zu sichern (man beachte, dass Fehlen eines entsprechenden POP ECX im Funktionsepilog). Tatsächlich reserviert der Befehl 4 Byte auf dem Stack, um die Variable x speichern zu können.

Auf x wird mithilfe des _x$ Makros (es entspricht -4) und des EBP Registers, das auf den aktuellen Stack Frame zeigt, zugegriffen. Während der Dauer der Funktionsausführung zeigt EBP auf den aktuellen Stack Frame, wodurch mittels EBP+offset auf lokalen Variablen und Funktionsargumente zugegriffen werden kann.

x is to be accessed with the assistance of the _x$ macro (it equals to -4) and the EBP register pointing to the current frame.

Es ist auch möglich, das ESP Register zu diesem Zweck zu verwenden, aber dies ist ungebräuchlich, da es sich häufig verändert. Der Wert von EBP kann als eingefrorener Wert des Wertes von ESP zu Beginn der Funktionsausführung verstanden werden.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a *frozen state* of the value in ESP at the start of the function's execution.

Hier ist ein typisches Layour eines Stack Frames in einer 32-Bit-Umgebung:

…	…
EBP-8	local variable #2, in IDA gekennzeichnet als var_8
EBP-4	local variable #1, in IDA gekennzeichnet als var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	Argument#1, in IDA gekennzeichnet als arg_0
EBP+0xC	Argument#2, in IDA gekennzeichnet als arg_4
EBP+0x10	Argument#3, in IDA gekennzeichnet als arg_8
…	…

Die Funktion scanf() in unserem Beispiel hat zwei Argumente.

Das erste ist ein Pointer auf den String %d und das zweite ist die Adresse der Variablen x.

Zunächst wird die Adresse der Variablen x durch den Befehl
lea eax, DWORD PTR _x$[ebp] in das EAX Register geladen.

LEA steht für *load effective address* und wird häufig benutzt, um eine Adresse zu erstellen (**??** on page ??). In diesem Fall speichert LEA einfach die Summe des EBP Registers und des _$ Makros im Register EAX. Dies entspricht dem Befehl lea eax, [ebp-4].

Es wird also 4 von Wert in EBP abgezogen und das Ergebnis in das Register EAX geladen. Danach wird der Wert in EAX auf dem Stack abgelegt und scanf() wird aufgerufen.

Anschließend wird printf() mit einem Argument aufgerufen–einen Pointer auf den String: You entered %d...\n.

Das zweite Argument wird mit `mov ecx, [ebp-4]` vorbereitet. Dieser Befehl speichert den Wert der Variablen x (nicht seine Adresse) im Register ECX.

Schließlich wird der Wert in ECX auf dem Stack gespeichert und das letzte `printf()` wird aufgerufen.

MSVC + OllyDbg

Schauen wir uns diese Beispiel in OllyDbg an. Wir laden es und drücken F8 (step over) bis wir unsere ausführbare Datei anstelle von `ntdll.dll` erreicht haben. Wir scrollen nach oben bis `main()` erscheint.

Wir klicken auf den ersten Befehl (PUSH EBO), drücken F2 (*set a breakpoint*), dann F9 (*Run*). Der Breakpoint wird ausgelöst, wenn die Funktion `main()` beginnt.

Verfolgen wir den Ablauf bis zu der Stelle, an der die Adresse der Variablen x berechnet wird:

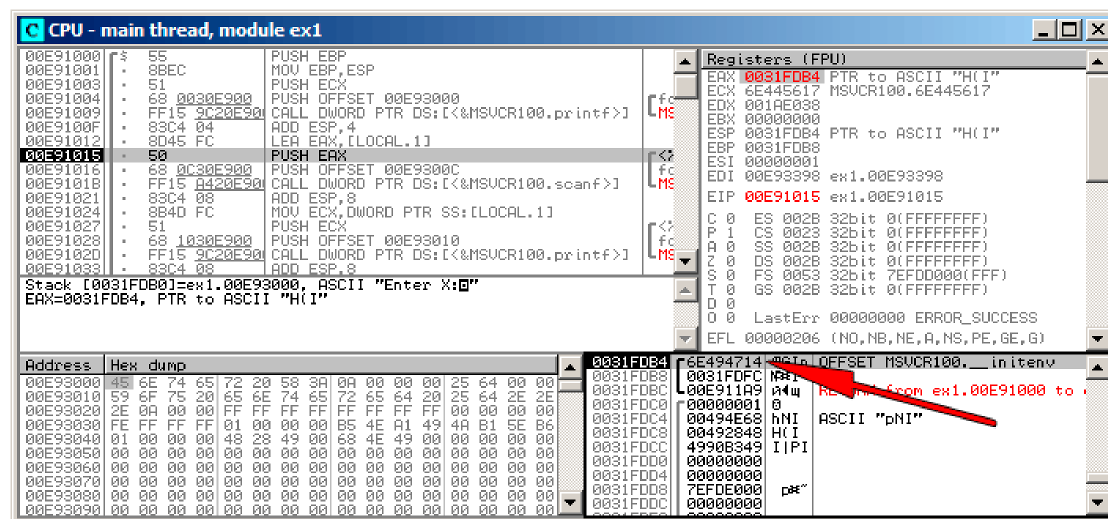


Abbildung 1.13: OllyDbg: Die Adresse der lokalen Variable wird berechnet.

Wir machen einen Rechtsklick auf EAX in Registerfenster und wählen „Follow in stack".

Diese Adresse wird im Stackfenster erscheinen. Der rote Pfeil wurde nachträglich hinzugefügt; er zeigt auf die Variable im lokalen Stack. Im Moment enthält diese Speicherstelle Zufallswerte (0x6E494714). Jetzt wird mithilfe des PUSH Befehls die Adresse dieses Stackelements auf demselben Stack an der folgenden Position gespeichert. Verfolgen wir den Ablauf mit F8 bis die Ausführung von `scanf()` abgeschlossen ist. Während der Ausführung von `scanf()` geben wir beispielsweise 123 in der Konsole ein:

```
Enter X:
123
```

scanf() ist bereits beendet:

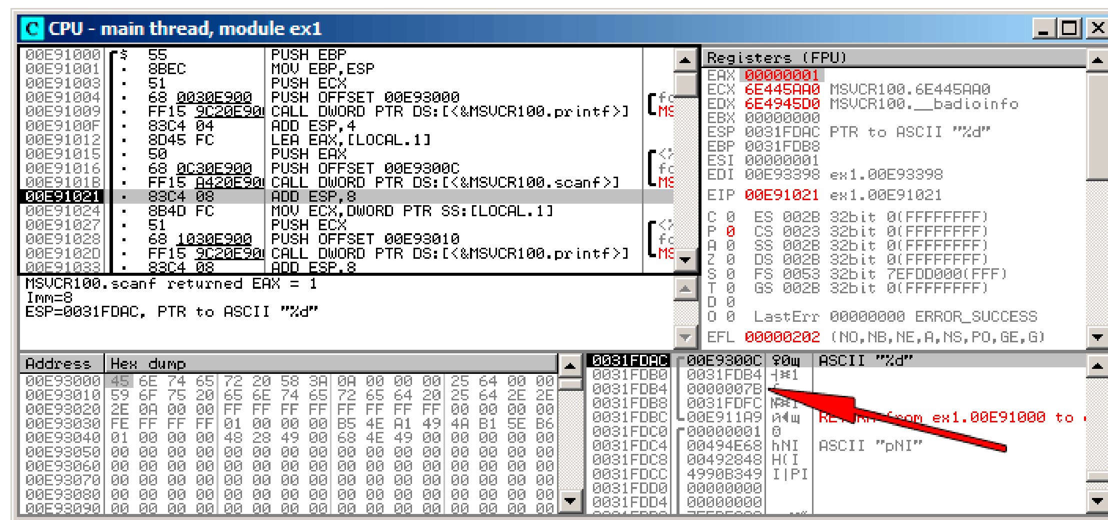


Abbildung 1.14: OllyDbg: scanf() wurde ausgeführt

scanf() liefert 1 im EAX Register zurück, was aussagt, dass die Funktion einen Wert erfolgreich eingelesen hat. Wenn wir wiederum auf das zugehörige Stackelement für die lokale Variable schauen, enthält diese nun den Wert 0x7B (dez. 123).

Im weiteren Verlauf wird dieser Wert vom Stack in das ECX Register kopiert und an printf() übergeben:

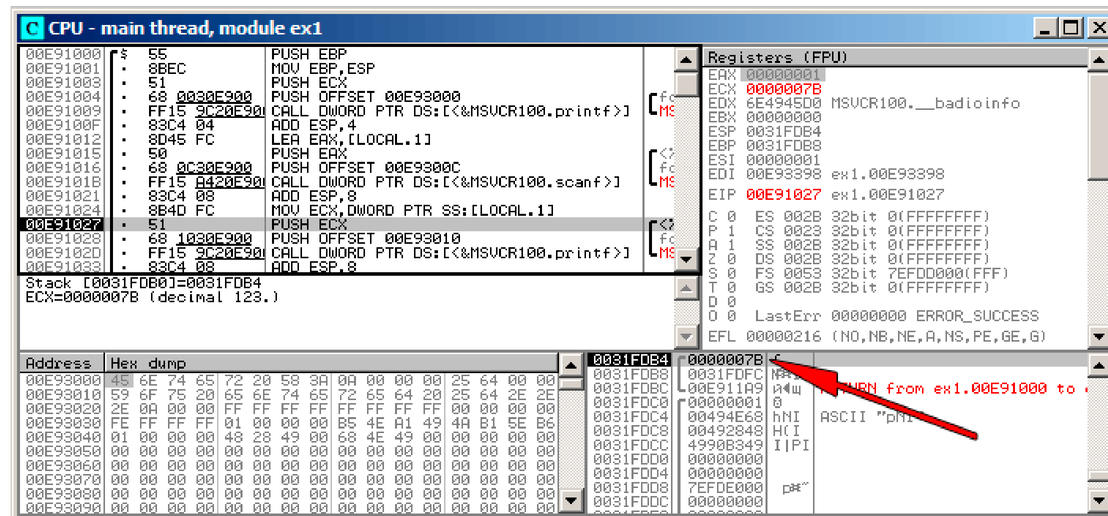


Abbildung 1.15: OllyDbg: Wert für Übergabe an printf() vorbereiten.

GCC

Kompilieren wir diesen Code mit GCC 4.4.1 unter Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
```

```
                leave
                retn
main            endp
```

GCC ersetzt den Aufruf von `printf()` durch einen Aufruf von `puts()`. Der Grund hierfür wurde bereits in (1.5.3 on page 28) erklärt.

Genau wie im MSVC Beispiel werden die Argumente mithilfe des Befehls `MOV` auf dem Stack abgelegt.

By the way

Dieses einfache Beispiel ist übrigens eine Demonstration der Tatsache, dass der Compiler eine Liste von Ausdrücken in einem C/C++-Block in eine sequentielle Liste von Befehlen übersetzt. Es gibt nichts zwischen zwei C/C++-Anweisungen und genauso verhält es sich auch im Maschinencode. Der Control Flow geht von einem Ausdruck direkt an den folgenden über.

x64

Hier zeigt sich ein ähnliches Bild mit dem Unterschied, dass die Register anstelle des Stacks für die Übergabe der Funktionsargumente verwendet werden.

MSVC

Listing 1.49: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
```

```
main    ENDP
_TEXT   ENDS
```

GCC

Listing 1.50: Optimierender GCC 4.4.6 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 24
        ret
```

ARM

Optimierender Keil 6/2013 (Thumb Modus)

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8           = -8
.text:00000042
.text:00000042 08 B5        PUSH    {R3,LR}
.text:00000044 A9 A0        ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000046 06 F0 D3 F8  BL      __2printf
.text:0000004A 69 46        MOV     R1, SP
.text:0000004C AA A0        ADR     R0, aD  ; "%d"
.text:0000004E 06 F0 CD F8  BL      __0scanf
.text:00000052 00 99        LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0        ADR     R0, aYouEnteredD___ ; "You entered
    %d...\n"
.text:00000056 06 F0 CB F8  BL      __2printf
```

```
.text:0000005A 00 20        MOVS     R0, #0
.text:0000005C 08 BD        POP      {R3,PC}
```

Damit `scanf()` Elemente einlesen kann, benötigt die Funktion einen Paramter–einen Pointer vom Typ *int*. *int* hat die Größe 32 Bit, wir benötigen also 4 Byte, um den Wert im Speicher abzulegen, und passt daher genau in ein 32-Bit-Register. Auf dem Stack wird Platz für die lokalen Variable x reserviert und IDA bezeichnet diese Variable mit *var_8*. Eigentlich ist aber an dieser Stelle gar nicht notwendig, Platz auf dem Stack zu reservieren, da **SP!** (Stapel-Zeiger bereits auf die Adresse zeigt und auch direkt verwendet werden kann.

Der Wert von **SP!** wird also in das R1 Register kopiert und zusammen mit dem Formatierungsstring an `scanf()` übergeben.

Später wird mithilfe des LDR Befehls dieser Wert vom Stack in das R1 Register verschoben um an `printf()` übergeben werden zu können.

ARM64

Listing 1.51: Nicht optimierender GCC 4.9.1 ARM64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
scanf_main:
; subtrahiere 32 von SP, speichere dann FP und LR im Stack Frame:
        stp     x29, x30, [sp, -32]!
; setze Stack Frame (FP=SP)
        add     x29, sp, 0
; lade Pointer auf den "Enter X:" String:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; X0=Pointer auf den "Enter X:" String
; print it:
        bl      puts
; lade Pointer auf den "%d" String:
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; finde Platz im Stack Frame für die Variable "x" (X1=FP+28):
        add     x1, x29, 28
; X1=Adresse der Variablen "x"
; übergebe die Adresse and scanf() und rufe auf:
        bl      __isoc99_scanf
; lade 32-Bit-Wert aus der Variable in den Stack Frame:
        ldr     w1, [x29,28]
; W1=x
; lade Pointer auf den "You entered %d...\n" String
; printf() nimmt den Textstring aus X0 und die Variable "x" aus X1 (oder W1)
        adrp    x0, .LC2
```

```
32          add     x0, x0, :lo12:.LC2
33          bl      printf
34  ; return 0
35          mov     w0, 0
36  ; stelle FP und LR wieder her, addiere dann 32 zu SP:
37          ldp     x29, x30, [sp], 32
38          ret
```

Im Stack Frame werden 32 Byte reserviert, was deutlich mehr als benötigt ist. Vielleicht handelt es sich um eine Frage des Aligning (dt. Angleichens) von Speicheradressen. Der interessanteste Teil ist, im Stack Frame einen Platz für die Variable x zu finden (Zeile 22). Warum 28? Irgendwie hat der Compiler entschieden die Variable am Ende des Stack Frames anstatt an dessen Beginn abzulegen. Die Adresse wird an `scanf()` übergeben; diese Funktion speichert den Userinput an der genannten Adresse im Speicher. Es handelt sich hier um einen 32-Bit-Wert vom Typ *int*. Der Wert wird in Zeile 27 abgeholt und dann an `printf()` übergeben.

MIPS

Auf dem lokalen Stack wird Platz für die Variable x reserviert und als $\$sp+24$ referenziert.

Die Adresse wird an `scanf()` übergeben und der Userinput wird mithilfe des Befehls `LW` („Load Word“) geladen und dann an `printf()` übergeben.

Listing 1.52: Optimierender GCC 4.4.5 (Assemblercode)

```
$LC0:
        .ascii  "Enter X:\000"
$LC1:
        .ascii  "%d\000"
$LC2:
        .ascii  "You entered %d...\012\000"
main:
; Funktionsprolog:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-40
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,36($sp)
; Aufruf von puts():
        lw      $25,%call16(puts)($28)
        lui     $4,%hi($LC0)
        jalr    $25
        addiu   $4,$4,%lo($LC0) ; branch delay slot
; Aufruf von scanf():
        lw      $28,16($sp)
        lui     $4,%hi($LC1)
        lw      $25,%call16(__isoc99_scanf)($28)
; setze 2. Argument von scanf(), $a1=$sp+24:
        addiu   $5,$sp,24
        jalr    $25
        addiu   $4,$4,%lo($LC1) ; branch delay slot
```

```
; Aufruf von printf():
        lw      $28,16($sp)
; setze 2. Argument von printf(),
; lade Wort nach Adresse $sp+24:
        lw      $5,24($sp)
        lw      $25,%call16(printf)($28)
        lui     $4,%hi($LC2)
        jalr    $25
        addiu   $4,$4,%lo($LC2) ; branch delay slot

; Funktionsepilog:
        lw      $31,36($sp)
; setze Rückgabewert auf 0:
        move    $2,$0
; return:
        j       $31
        addiu   $sp,$sp,40      ; branch delay slot
```

IDA stellt das Stack Layout wie folgt dar:

Listing 1.53: Optimierender GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_18   = -0x18
.text:00000000 var_10   = -0x10
.text:00000000 var_4    = -4
.text:00000000
; Funktionsprolog:
.text:00000000          lui     $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x28
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x28+var_4($sp)
.text:00000010          sw      $gp, 0x28+var_18($sp)
; Aufruf von puts():
.text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018          lui     $a0, ($LC0 >> 16)  # "Enter X:"
.text:0000001C          jalr    $t9
.text:00000020          la      $a0, ($LC0 & 0xFFFF)  # "Enter X:" ; branch
    delay slot
; Aufruf von scanf():
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16)  # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; setze 2. Argument von scanf(), $a1=$sp+24:
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9  ; branch delay slot
.text:00000038          la      $a0, ($LC1 & 0xFFFF)  # "%d"
; Aufruf von printf():
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; setze 2. Argument von printf(),
; lade Word nach Adresse $sp+24:
```

```
.text:00000040          lw      $a1, 0x28+var_10($sp)
.text:00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048          lui     $a0, ($LC2 >> 16)  # "You entered %d...\n"
.text:0000004C          jalr    $t9
.text:00000050          la      $a0, ($LC2 & 0xFFFF)  # "You entered %d...\n"
    ; branch delay slot
; Funktionsepilog:
.text:00000054          lw      $ra, 0x28+var_4($sp)
; setze Rückgabewert auf 0:
.text:00000058          move    $v0, $zero
; return:
.text:0000005C          jr      $ra
.text:00000060          addiu   $sp, 0x28 ; branch delay slot
```

1.9.2 Häufiger Fehler

Ein häufiger (Tipp)-Fehler besteht darin, den Wert von *x* anstatt eines Pointers auf *x* zu übergeben.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", x); // BUG

        printf ("You entered %d...\n", x);

        return 0;
};
```

Was geschieht hier? *x* ist nicht uninitialisiert und enthält Zufallswerte vom lokalen Stack. Wenn `scanf()` aufgerufen wird, nimmt es den eingegebenen String vom Benutzer, wandelt ihn in eine Zahl um und versucht ihn nach *x* zu schreiben, wobei *x* wie eine Speicheradresse behandelt wird. Aber hier liegen Zufallswerte vor, sodass `scanf()` versucht an eine zufällige Speicherstelle zu schreiben. Höchstwahrscheinlich wird der Prozess dadurch abstürzen.

Bemerkenswert ist, dass manche CRT-Bibliotheken im Debug Build gut erkennbare Muster in den gerade reservierten Speicher schreiben, wie z.B. 0xCCCCCCCC oder 0x0BADF00D usw. In diesem Fall könnte *x* den Wert 0xCCCCCCCC enthalten und `scanf()` würde versuchen in die Adresse 0xCCCCCCCC zu schreiben. Wenn man nun bemerkt, dass irgendein Code im Prozess in die Adresse 0xCCCCCCCC schreiben möchte, weiß man, dass eine uninitialisierte Variable (oder ein Pointer) verwendet werden. Dies ist besser als wenn der frisch reservierte Speicher einfach gelöscht würde.

1.9.3 Globale Variablen

Was passiert, wenn die x-Variable aus dem letzten Beispiel nicht lokal sondern global ist? In dem Fall wäre sie von jeder Stelle aus zugreifbar, nicht nur aus dem Funktions-Rumpf. Globale Variablen gelten als Anti-Pattern, aber für den Lerneffekt können wir folgendes tun:

```
#include <stdio.h>

// x ist jetzt eine globale Variable
int x;

int main()
{
    printf ("Wert für x:\n");

    scanf ("%d", &x);

    printf ("Sie haben %d... eingegeben\n", x);

    return 0;
};
```

MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
```

```
    ret     0
_main    ENDP
_TEXT    ENDS
```

In diesem Falle wird die Variable x im _DATA Segment definiert und kein Speicher auf dem lokalen Stack wird reserviert. Es erfolgt ein direkter Zugriff ohne Umweg über den Stack. Nicht initialisierte globale Variablen verbrauchen keinen Platz in der ausführbaren Datei (und wirklich: warum sollte man Speicher reservieren, wenn die Variable auf Anfang auf 0 gesetzt wird?), aber wenn jemand auf ihre Adresse zugreift, wird das BS einen Block Nullen reservieren[66].

Weisen wir nun der Variablen ausdrücklich einen Wert zu:

```
int x=10; // Defaultwert
```

Wir erhalten:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Wir sehen hier einen Wert 0xA vom Typ DWORD (DD steht für DWORD = 32 Bit) für diese Variable.

Wenn man die kompilierte .exe in IDA öffnet, sieht man, dass die Variable x am Anfang des _DATA Segments angelegt wird und danach sehen wir Textstrings.

Here we see a value 0xA of DWORD type (DD stands for DWORD = 32 bit) for this variable.

Wenn man die kompilierte .exe aus dem vorangegangenen Beispiel in IDA öffnet, in dem der Wert von x nicht gesetzt wurde, sieht man etwa das Folgende:

Listing 1.54: IDA

```
.data:0040FA80 _x                dd ?    ; DATA XREF: _main+10
.data:0040FA80                           ; _main+22
.data:0040FA84 dword_40FA84      dd ?    ; DATA XREF: _memset+1E
.data:0040FA84                           ; unknown_libname_1+28
.data:0040FA88 dword_40FA88      dd ?    ; DATA XREF: ___sbh_find_block+5
.data:0040FA88                           ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem             dd ?    ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C                           ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90      dd ?    ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90                           ; __calloc_impl+72
.data:0040FA94 dword_40FA94      dd ?    ; DATA XREF: ___sbh_free_block+2FE
```

_x ist mit ? markiert zusammen mit dem Rest der Variablen, die nicht initialisiert werden müssen. Daraus folgt, dass nachdem die .exe in den Speicher geladen wurde, Platz für alle diese Variablen angelegt und mit Nullen gefüllt werden muss [*ISO/IEC*

[66]That is how a VM behaves

9899:TC3 (C C99 standard), (2007)6.7.8p10]. Aber in der .exe Datei selbst, belegen diese nicht initialisierten Variablen keinerlei Platz. Dies ist beispielsweise für große Arrays üblich.

MSVC: x86 + OllyDbg

Hier sehen die Dinge noch einfacher aus:

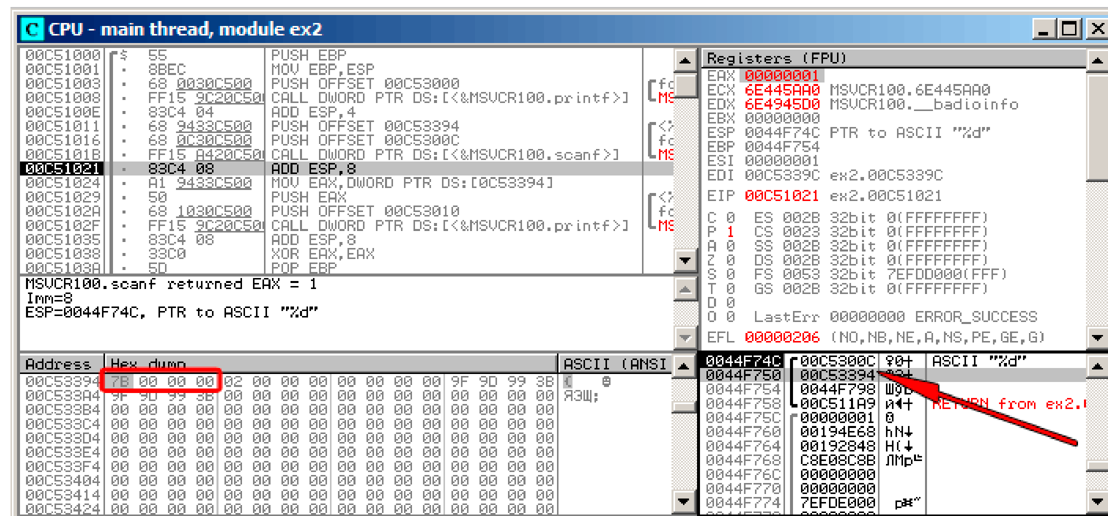


Abbildung 1.16: OllyDbg: nach Ausführung von `scanf()`

Die Variable befindet sich im Datensegment. Nachdem der PUSH Befehl (der die Adresse von x speichert) ausgeführt worden ist, erscheint die Adresse im Stackfenster. Wir machen einen Rechtsklick auf die Zeile und wählen „Follow in dump". Die Variable erscheint nun im Speicherfenster auf der linken Seite. Nachdem wir in der Konsole 123 eingegeben haben, erscheint `0x7B` im Speicherfenster (siehe markiertes Feld im Screenshot).

Warum ist das erste Byte 7B? Logisch gedacht müsste dort `00 00 00 7B` sein. Der Grund dafür ist die sogenannte Endianess und x86 verwendet *litte Endian*. Dies bedeutet, dass das niederwertigste Byte zuerst und das höchstwertigste zuletzt geschrieben werden. Für mehr Informationen dazu siehe: **??** on page ??. Zurück zu Beispiel: der 32-Bit-Wert wird von dieser Speicheradresse nach EAX geladen und an `printf()` übergeben.

Die Speicheradresse von x ist `0x00C53394`.

In OllyDbg können wir die Speicherzuordnung des Prozesses nachvollziehen (Alt-M) und wir erkennen, dass sich diese Adresse innerhalb des `.data` PE-Segments von unserem Programm befindet:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00067000				Map	R	R	C:\Windows\System32\locale.nl
00190000	00005000			Heap	Priv	RW	RW	
00209000	00007000				Priv	RW Gua	RW Gua	
0044C000	00001000				Priv	RW Gua	RW Gua	
0044D000	00003000			Stack of main thread	Priv	RW	RW	
00590000	00007000				Priv	RW	RW	
00750000	0000C000			Default heap	Priv	RW	RW	
00C50000	00001000	ex2		PE header	Img	R	RWE Cop	
00C51000	00001000	ex2	.text	Code	Img	R E	RWE Cop	
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE Cop	
00C53000	00001000	ex2	.data	Data	Img	RW	RWE Cop	
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE Cop	
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE Cop	
6E3E1000	000B2000	MSVCR100	.text	Code,imports,exports	Img	R E	RWE Cop	
6E493000	00006000	MSVCR100	.data	Data	Img	RW Cop	RWE Cop	
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE Cop	
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop	
755D1000	00003000				Img	R E	RWE Cop	
755D4000	00001000				Img	RW	RWE Cop	
755D5000	00003000				Img	R	RWE Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop	
755E1000	0004D000				Img	R E	RWE Cop	
7562E000	00005000				Img	RW Cop	RWE Cop	
75633000	00009000				Img	R	RWE Cop	
75640000	00001000	Mod_7564		PE header	Img	R	RWE Cop	
75641000	00038000				Img	R E	RWE Cop	
75679000	00002000				Img	RW	RWE Cop	
7567B000	00004000				Img	R	RWE Cop	
76F50000	00010000	kernel32		PE header	Img	R	RWE Cop	
76F60000	000D0000	kernel32	.text	Code,imports,exports	Img	R E	RWE Cop	
77030000	00010000	kernel32	.data	Data	Img	RW Cop	RWE Cop	
77040000	00010000	kernel32	.rsrc	Resources	Img	R	RWE Cop	
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE Cop	
77810000	00001000	KERNELBASE		PE header	Img	R	RWE Cop	
77811000	00040000	KERNELBASE	.text	Code,imports,exports	Img	R E	RWE Cop	
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE Cop	
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE Cop	
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE Cop	
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE Cop	
77B21000	00102000				Img	R E	RWE Cop	
77C23000	0002F000				Img	R	RWE Cop	
77C52000	0000C000				Img	RW Cop	RWE Cop	
77C5E000	0006B000				Img	R	RWE Cop	
77D00000	00001000	ntdll		PE header	Img	R	RWE Cop	
77D10000	000D6000	ntdll	.text	Code,exports	Img	R E	RWE Cop	
77DF0000	00001000	ntdll	RT	Code	Img	R E	RWE Cop	
77E00000	00009000	ntdll	.data	Data	Img	RW Cop	RWE Cop	

Abbildung 1.17: OllyDbg: Speicherzuordnung

GCC: x86

Unter Linux zeigt sich ein ähnliches Bild, mit dem Unterschied, dass die nicht initialisierten Variablen im `_bss` Segment gehalten werden. I der ELF[67] Datei hat dieses Segment die folgenden Attribute:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Wenn man nun der Variablen einen Wert zuweist, z.B. 10, muss diese im `_data` Segment abgelegt werden, dass die folgenden Attribute aufweist:

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

[67] Executable and Linkable Format: In *NIX Systemen einschließlich Linux weit verbreitetes Format für ausführbare Dateien

MSVC: x64

Listing 1.55: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Der Code ist beinahe der gleich wie in x86. Man beachte, dass die Adresse der Variable x mittels des Befehls LEA an die Funktion `scanf()` übergeben wird, wohingegen der Wert der Variablen an das zweite `printf()` mit einem `MOV` Befehl übergeben wird. `DWORD PTR`—ist ein Teil der Assemblersprache (mit keinerlei Verbindung zum Maschinencode) und zeigt an, dass die Variablengröße 32 Bit beträgt und der `MOV` Befehl entsprechend aufgebaut sein muss.

ARM: Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.56: IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000         AREA .text, CODE
...
.text:00000000 main
.text:00000000         PUSH    {R4,LR}
.text:00000002         ADR     R0, aEnterX  ; "Enter X:\n"
.text:00000004         BL      __2printf
.text:00000008         LDR     R1, =x
.text:0000000A         ADR     R0, aD       ; "%d"
```

```
.text:0000000C          BL      __0scanf
.text:00000010          LDR     R0, =x
.text:00000012          LDR     R1, [R0]
.text:00000014          ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016          BL      __2printf
.text:0000001A          MOVS    R0, #0
.text:0000001C          POP     {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A          DCB    0
.text:0000002B          DCB    0
.text:0000002C off_2C   DCD x                   ; DATA XREF: main+8
.text:0000002C                                  ; main+10
.text:00000030 aD       DCB "%d",0              ; DATA XREF: main+A
.text:00000033          DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; DATA XREF:
    main+14
.text:00000047          DCB 0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048          AREA .data, DATA
.data:00000048          ; ORG 0x48
.data:00000048          EXPORT x
.data:00000048 x        DCD 0xA                 ; DATA XREF: main+8
.data:00000048                                  ; main+10
.data:00000048 ; .data ends
```

Nun ist x eine globale Variable und aus diesem Grund in einem anderen Segment, nämlich dem Datensegment (*.data*) angelegt. Man könnte sich fragen, warum die Textstrings sich im Codesegment (*.text*) befindet und x nicht. Das liegt daran, dass x eine Variable ist und sich damit per definitionem ihr Wert ändern kann; mehr noch, er kann sich möglicherweise oft ändern. Textstrings dagegen sind Konstanten, können also nicht geändert werden, und befinden sich deshalb im *.text* Segment. Das Codesegment kann manchmal in einem **ROM!**[68] Chip liegen (erinnern wir uns, dass wir es hier mit eingebetteter Mikroelektronik zu tun haben und Speicherknappheit ein häufiges Problem ist) und änderbare Variablen —im RAM[69].

Es ist nicht besonders ökonomisch konstante Variablen im RAM zu speichern, wenn ein ROM verfügbar ist.

Mehr noch, Konstanten im RAM müssen initialisiert werden, denn nach dem Einschalten enthält der RAM naturgemäß zufällige Informationen.

Im Folgenden sehen wir einen Pointer auf die Variable x (`off_2C`) im Codesegment und bemerken, dass alle Operationen auf der Variable über den Pointer laufen.

Dies liegt daran, dass die Variable x auch an einem weit entfernten Codefragment liegen könnte, weshalb ihre Adresse irgendwo in der Nähe des Codes gespeichert werden muss.

[68]**ROM!**
[69]Random-Access Memory

Der Befehl LDR im Thumb Mode kann nur Variablen in einer Reichweite von 1020 Bytes von seiner Speicherstelle adressieren und im ARM Mode —beträgt die Spannweite der Variablen ± 4095 Bytes.

Also muss die Adresse der Variable x sich in der Nähe befinden, denn es gibt keine Garantie, dass der Linker in der Lage ist, die Variable in der Nähe des Codes zu platzieren, sie könnte sich sogar auf einem externen Memory Chip befinden.

Eine weitere Sache: wenn eine Variable as *konstant* deklariert wird, legt der Keil Compiler sie im `.constdata` Segment ab.

Möglicherweise könnte der Linker dieses Segment anschließend gemeinsam mit dem Codesegment im ROM ablegen.

ARM64

Listing 1.57: Nicht optimierender GCC 4.9.1 ARM64

```
        .comm   x,4,4
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
f5:
; speichere FP und LR im Stack Frame:
        stp     x29, x30, [sp, -16]!
; setze Stack Frame (FP=SP)
        add     x29, sp, 0
; lade Pointer auf den "Enter X:" String:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; lade Pointer auf den "%d" String:
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; bilde Adresse der globalen Variable x:
        adrp    x1, x
        add     x1, x1, :lo12:x
        bl      __isoc99_scanf
; bilde Adress der globalen Variable x erneut:
        adrp    x0, x
        add     x0, x0, :lo12:x
; lade Wert aus dem Speicher an dieser Adresse:
        ldr     w1, [x0]
; lade Pointer auf den "You entered %d...\n" string:
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; return 0
        mov     w0, 0
; stelle FP und LR wieder her:
        ldp     x29, x30, [sp], 16
```

```
37        ret
```

In diesem Fall ist die Variable x als global deklariert und ihre Adresse wird mithilfe des Befehlspaares ADRP/ADD berechnet (Zeilen 21 und 25).

MIPS

Nicht initialisierte globale Variable

Nun ist x eine globale Variable. Wir kompilieren zu einer ausführbaren Datei anstelle eines Objectfiles und laden diese in IDA. IDA stellt die Variable x in der ELF Section *.sbss* dar (erinnern Sie sich an „Global Pointer“?1.5.4 on page 33), denn die Variable ist zu Beginn nicht initialisiert.

Listing 1.58: Optimierender GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10   = -0x10
.text:004006C0 var_4    = -4
.text:004006C0
; Funktionsprolog:
.text:004006C0          lui     $gp, 0x42
.text:004006C4          addiu   $sp, -0x20
.text:004006C8          li      $gp, 0x418940
.text:004006CC          sw      $ra, 0x20+var_4($sp)
.text:004006D0          sw      $gp, 0x20+var_10($sp)
; Aufruf von puts():
.text:004006D4          la      $t9, puts
.text:004006D8          lui     $a0, 0x40
.text:004006DC          jalr    $t9 ; puts
.text:004006E0          la      $a0, aEnterX     # "Enter X:" ; branch delay
    slot
; Aufruf von scanf():
.text:004006E4          lw      $gp, 0x20+var_10($sp)
.text:004006E8          lui     $a0, 0x40
.text:004006EC          la      $t9, __isoc99_scanf
; bereite Adresse von x vor:
.text:004006F0          la      $a1, x
.text:004006F4          jalr    $t9 ; __isoc99_scanf
.text:004006F8          la      $a0, aD          # "%d" ; branch delay slot
; Aufruf von printf():
.text:004006FC          lw      $gp, 0x20+var_10($sp)
.text:00400700          lui     $a0, 0x40
; hole Adresse von x:
.text:00400704          la      $v0, x
.text:00400708          la      $t9, printf
; lade Wert von "x" und übergib diesen an printf() in $a1:
.text:0040070C          lw      $a1, (x - 0x41099C)($v0)
.text:00400710          jalr    $t9 ; printf
.text:00400714          la      $a0, aYouEnteredD___  # "You entered %d...\n"
    ; branch delay slot
; Funktionsepilog:
.text:00400718          lw      $ra, 0x20+var_4($sp)
.text:0040071C          move    $v0, $zero
```

```
.text:00400720          jr      $ra
.text:00400724          addiu   $sp, 0x20  ; branch delay slot

...

.sbss:0041099C  # Segment type: Uninitialized
.sbss:0041099C          .sbss
.sbss:0041099C          .globl x
.sbss:0041099C x:       .space 4
.sbss:0041099C
```

IDA reduziert die Anzahl der Informationen; also erzeugen wir auch ein Listing mit objdump und kommentieren es:

Listing 1.59: Optimierender GCC 4.4.5 (objdump)

```
004006c0 <main>:
; Funktionsprolog:
  4006c0:  3c1c0042  lui     gp,0x42
  4006c4:  27bdffe0  addiu   sp,sp,-32
  4006c8:  279c8940  addiu   gp,gp,-30400
  4006cc:  afbf001c  sw      ra,28(sp)
  4006d0:  afbc0010  sw      gp,16(sp)
; Aufruf von puts():
  4006d4:  8f998034  lw      t9,-32716(gp)
  4006d8:  3c040040  lui     a0,0x40
  4006dc:  0320f809  jalr    t9
  4006e0:  248408f0  addiu   a0,a0,2288 ; branch delay slot
; Aufruf von scanf():
  4006e4:  8fbc0010  lw      gp,16(sp)
  4006e8:  3c040040  lui     a0,0x40
  4006ec:  8f998038  lw      t9,-32712(gp)
; bereite von Adresse von x vor:
  4006f0:  8f858044  lw      a1,-32700(gp)
  4006f4:  0320f809  jalr    t9
  4006f8:  248408fc  addiu   a0,a0,2300 ; branch delay slot
; Aufruf von printf():
  4006fc:  8fbc0010  lw      gp,16(sp)
  400700:  3c040040  lui     a0,0x40
; hole Adresse von x:
  400704:  8f828044  lw      v0,-32700(gp)
  400708:  8f99803c  lw      t9,-32708(gp)
; lade Wert von "x" und übergib diesen an printf() in $a1:
  40070c:  8c450000  lw      a1,0(v0)
  400710:  0320f809  jalr    t9
  400714:  24840900  addiu   a0,a0,2304 ; branch delay slot
; Funktionsepilog:
  400718:  8fbf001c  lw      ra,28(sp)
  40071c:  00001021  move    v0,zero
  400720:  03e00008  jr      ra
  400724:  27bd0020  addiu   sp,sp,32   ; branch delay slot
; einige NOPs um Beginn der nächsten Funktion auf 16-Byte-Grenze zu legen:
  400728:  00200825  move    at,at
  40072c:  00200825  move    at,at
```

Wir erkennen, dass die Adresse der Variablen x mit GP aus einem 64KiB Datenpuffer gelesen wird und ein negativer Offset hinzugefügt wird (Zeile 18). Mehr noch, die Adressen der drei externen Funktionen, die in unserem Beispiel verwendet werden (`puts()`, `scanf()`, `printf()`) wwerden auch mithilfe von GP aus einem 64KiB globalen Datenpuffer gelesen (Zeile 9, 16 und 26). GP zeigt auf die Mitte des Puffers und ein solches Offset gibt an, dass die Adressen aller drei Funktionen genau wie die Adresse von x irgendwo am Anfang des Puffers gespeichert werden. Das ergibt Sinn, denn unser Beispiel ist winzig.

Eine andere bemerkenswerte Sache ist, dass die Funktion mit zwei NOPs (`MOVE $AT,$AT` — Befehle ohne Auswirking) schließt, damit der Beginn der nächsten Funktion auf eine 16-Byte-Grenze gelegt wird.

Initialisierte globale Variable

Verändern wir unser Beispiel dadurch, dass wir x einen Defaultwert geben:

```
int x=10; // Defaultwert
```

IDA zeigt hier, dass die Variable x im .data Segment liegt:

Listing 1.60: Optimierender GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8  = -8
.text:004006A0 var_4  = -4
.text:004006A0
.text:004006A0        lui     $gp, 0x42
.text:004006A4        addiu   $sp, -0x20
.text:004006A8        li      $gp, 0x418930
.text:004006AC        sw      $ra, 0x20+var_4($sp)
.text:004006B0        sw      $s0, 0x20+var_8($sp)
.text:004006B4        sw      $gp, 0x20+var_10($sp)
.text:004006B8        la      $t9, puts
.text:004006BC        lui     $a0, 0x40
.text:004006C0        jalr    $t9 ; puts
.text:004006C4        la      $a0, aEnterX     # "Enter X:"
.text:004006C8        lw      $gp, 0x20+var_10($sp)
; bereite höherwertigen Teil der Adresse von x vor:
.text:004006CC        lui     $s0, 0x41
.text:004006D0        la      $t9, __isoc99_scanf
.text:004006D4        lui     $a0, 0x40
; addiere niederwertigen Teil der Adresse von x:
.text:004006D8        addiu   $a1, $s0, (x - 0x410000)
; Adresse von x liegt nun in $a1.
.text:004006DC        jalr    $t9 ; __isoc99_scanf
.text:004006E0        la      $a0, aD          # "%d"
.text:004006E4        lw      $gp, 0x20+var_10($sp)
; hole ein Word aus dem Speicher:
.text:004006E8        lw      $a1, x
```

```
; Wert von x liegt nun in $a1.
.text:004006EC        la      $t9, printf
.text:004006F0        lui     $a0, 0x40
.text:004006F4        jalr    $t9 ; printf
.text:004006F8        la      $a0, aYouEnteredD___  # "You entered %d...\n"
.text:004006FC        lw      $ra, 0x20+var_4($sp)
.text:00400700        move    $v0, $zero
.text:00400704        lw      $s0, 0x20+var_8($sp)
.text:00400708        jr      $ra
.text:0040070C        addiu   $sp, 0x20

...

.data:00410920        .globl x
.data:00410920 x:     .word 0xA
```

Warum nicht .sdata? Vielleicht hängt es mit einer Option von GCC zusammen?

Nichtsdestotrotz befindet sich x jetzt in .data, also dem allgemeinen Speicherbereich, und wir können uns anschauen wie hier mit den Variablen gearbeitet wird.

Die Adresse der Variablen muss mithilfe eines Instruktionspaares gebildet werden. In unserem Fall sind dies LUI („Load Upper Immediate“) und ADDIU („Add Immediate Unsigned Word“).

Hier ist also das objdump Listing für eine genaue Untersuchung:

Listing 1.61: Optimierender GCC 4.4.5 (objdump)

```
004006a0 <main>:
  4006a0:  3c1c0042  lui     gp,0x42
  4006a4:  27bdffe0  addiu   sp,sp,-32
  4006a8:  279c8930  addiu   gp,gp,-30416
  4006ac:  afbf001c  sw      ra,28(sp)
  4006b0:  afb00018  sw      s0,24(sp)
  4006b4:  afbc0010  sw      gp,16(sp)
  4006b8:  8f998034  lw      t9,-32716(gp)
  4006bc:  3c040040  lui     a0,0x40
  4006c0:  0320f809  jalr    t9
  4006c4:  248408d0  addiu   a0,a0,2256
  4006c8:  8fbc0010  lw      gp,16(sp)
; bereite höherwertigen Teil der Adresse von x vor:
  4006cc:  3c100041  lui     s0,0x41
  4006d0:  8f998038  lw      t9,-32712(gp)
  4006d4:  3c040040  lui     a0,0x40
; addiere niederwertigen Teil der Adresse von x:
  4006d8:  26050920  addiu   a1,s0,2336
; Adresse von x liegt nun in $a1.
  4006dc:  0320f809  jalr    t9
  4006e0:  248408dc  addiu   a0,a0,2268
  4006e4:  8fbc0010  lw      gp,16(sp)
; höherwertiger Teil der Adresse von x liegt immer noch in $s0.
; addiere niederwertigen Teil und lade ein Word aus dem Speicher:
  4006e8:  8e050920  lw      a1,2336(s0)
; Wert von x liegt jetzt in $a1.
```

```
  4006ec:  8f99803c  lw      t9,-32708(gp)
  4006f0:  3c040040  lui     a0,0x40
  4006f4:  0320f809  jalr    t9
  4006f8:  248408e0  addiu   a0,a0,2272
  4006fc:  8fbf001c  lw      ra,28(sp)
  400700:  00001021  move    v0,zero
  400704:  8fb00018  lw      s0,24(sp)
  400708:  03e00008  jr      ra
  40070c:  27bd0020  addiu   sp,sp,32
```

Wir erkennen, dass die Adresse mit LUI und ADDIU gebildet wird, aber der höherwertige Teil der Adresse sich immer noch im $S0 Register befindet und es möglich ist, den Offset durch einen LW („Load Word") Befehl anzugeben, sodass ein einzelnes LW ausreicht um den Wert aus der Variablen zu lesen und an printf() zu übergeben.

Register, die temporäre Daten halten, beginnen mit T-, aber wir sehen hier auch einige, die mit dem Präfix S- beginnen. Die Inhalte dieser Register müssen gespeichert werden, bevor sie in anderen Funktionen verwendet werden können.

Das ist der Grund warum der Wert von $S0 an der Adresse 0x4006cc gesetzt und dann wieder an der Adresse 0x4006e8 verwendet wurde, nachdem scanf() aufgerufen wurde. Die Funktion scanf() verändert den Wert nicht.

1.9.4 scanf()

Wie bereits angemerkt ist es ein wenig altmodisch heutzutage noch scanf() zu verwenden. Wenn wir aber darauf angewiesen sind, müssen wir prüfen, ob scanf() ohne Fehler ausgeführt wurde.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

Gemäß dem Standard, gibt die Funktion scanf()[70] die Anzahl der erfolgreich gelesenen Argumente zurück.

In unserem Fall also 1, falls alles fehlerfrei funktioniert und der User eine Zahl eingibt oder im Fehlerfall (oder bei EOF[71]) — 0.

[70] scanf, wscanf: MSDN
[71] End of File (Dateiende)

Fügen wir ein wenig C Code hinzu, um den Rückgabewert von `scanf()` zu überprüfen und im Fehlerfall eine Fehlermeldung auszugeben.

Dies funktioniert wie erwartet:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

MSVC: x86

Wir erhalten den folgenden Assembleroutput (MSVC 2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

Die aufrufende Funktion (`main()`) benötigt das Ergebnis der aufgerufenen Funktion (`scanf()`), weshalb diese es über das Register EAX zurückgibt.

Wir prüfen mithilfe des Befehls `CMP EAX, 1` (*CoMPare*). Mit anderen Worten, wir vergleichen den Wert in EAX mit 1.

Ein bedingter `JNE` Sprung folgt auf den `CMP` Befehl. `JNE` steht für *Jump if Not Equal*.

Wenn also der Wert in EAX ungleich 1 ist, wird die CPU die Ausführung an der Stelle fortsetzen, die im Operanden von `JNE` steht, in unserem Fall `$LN2@main`. Den Control Flow an diese Adresse zu übergeben hat zur Folge, dass die Funktion `printf()` mit dem Argument `What you entered? Huh?` aufgerufen wird. Wenn aber alles funktioniert und der bedingte Sprung nicht ausgeführt wird, wird ein anderer Aufruf von

printf() mit zwei Argumenten ausgeführt:
'You entered %d...' und dem Wert von x.

Da in diesem Fall das zweite printf() nicht ausgeführt werden darf, befindet sich davor ein JMP (unbedingter Sprung). Dieser gibt den Control Flow ab an den Punkt nach dem zweiten printf(), genau vor dem XOR EAX EAX Befehl, welcher die Rückgabe von 0 implementiert.

Man kann also festhalten, dass der Vergleich von zwei Werten gewöhnlich durch ein CMP/Jcc Befehlspaar implementiert wird, wobei *cc* für *condition code*, also Sprungbedingung, steht. CMP vergleicht zwei Werte und setzt die Flags des Prozessors[72]. Jcc prüft diese Flags und entscheidet entweder den Control Flow an die angegebene Adresse zu übergeben oder nicht.

Es klingt möglicherweise paradox, aber der CMP Befehl ist tatsächlich ein SUB (subtract). Alle arithmetischen Befehle setzen die Flags des Prozessors, nicht nur CMP. Wenn wir 1 und 1 vergleichen, ist $1 - 1 = 0$ und daher wird das ZF Flag gesetzt (gleichbedeutend damit, dass das Ergebnis der letzten Berechnung 0 ergeben hat). ZF kann nur durch diesen Umstand gesetzt werden, nämlich, dass zwei Operanden gleich sind. JNE prüft nur das ZF Flag und springt nur, wenn dieses nicht gesetzt ist. JNE ist daher ein Synonym für JNZ (*Jump if Not Zero*). Der Assembler übersetzt JNE und JNZ in den gleichen Opcode. Der CMP Befehl kann also durch ein SUB ersetzt werden, aber mit dem Unterschied, dass SUB den Wert des ersten Operanden verändert. CMP bedeutet also *SUB ohne Speichern des Ergebnisses, aber mit Setzen der Flags*.

MSVC: x86: IDA

Es ist an der Zeit IDA auszuprobieren und etwas damit zu machen. Für Anfänger ist es übrigens eine gute Idee, die /MD Option in MSVC zu verwenden, da diese bewirkt, dass alle Standardfunktionen nicht mit der ausführbaren Datei verlinkt werden, sondern aus der Datei MSVCR*.DLL importiert werden. Dadurch ist es einfacher zu erkennen, welche Standardfunktionen verwendet werden und wo dies geschieht.

Bei der Codeanalyse in IDA ist es hilfreich Notizen für sich selbst (und andere) zu hinterlassen. Bei der Analyse dieses Beispiels sehen wir, dass JNZ im Falle eines Fehlers ausgeführt wird. Es ist nun möglich den Cursor auf das Label zu setzen, „n“ zu drücken und es in „error“ umzubenennen. Wir erstellen noch ein Label—in „exit“. Hier ist das Ergebnis:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X:\n"
.text:00401009       call    ds:printf
```

[72] zu x86 Flags, siehe auch: wikipedia.

```
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

So ist es etwas einfacher den Code zu verstehen. Natürlich ist es aber auch keine gute Idee, jeden Befehl zu kommentieren.

Man kann Teile einer Funktion in IDA auch einklappen. Um dies zu tun, markiert man den Block und drückt dann Ctrl-„–" auf dem Zahlenblock der Tastatur und gibt den stattdessen anzuzeigenden Text ein.

Verstecken wir zwei Blöcke und geben ihnen Namen:

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000       assume cs:_text
.text:00401000       ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024       cmp  eax, 1
.text:00401027       jnz  short error
.text:00401029 ; print result
.text:0040103B       jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push offset aWhat ; "What you entered? Huh?\n"
.text:00401042       call ds:printf
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
```

```
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

Um eingeklappte Teile des Code wieder auszuklappen, verwendet man Ctrl-„+“ auf dem Zahlenblock der Tastatur.

Durch Drücken von der „Leertaste“ sehen wir, wie IDA die Funktion als Graph darstellt:

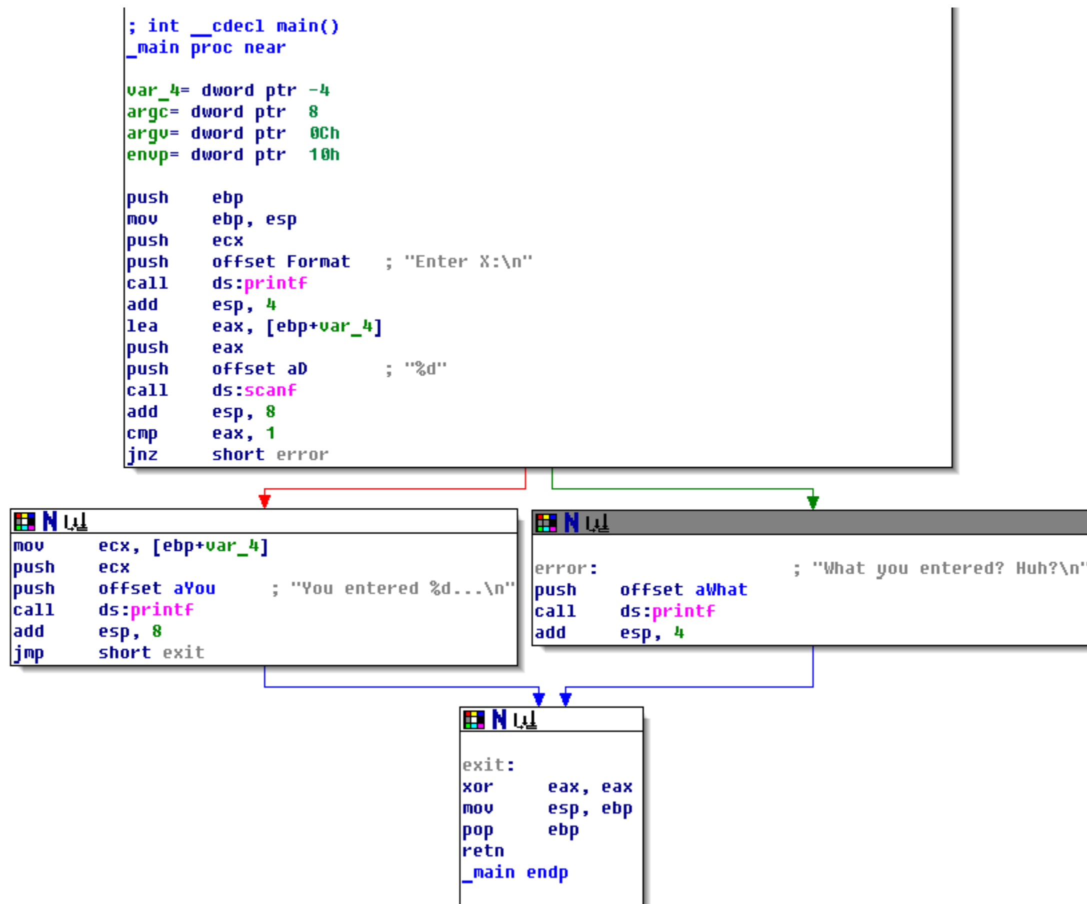


Abbildung 1.18: Graph Modus in IDA

Es gibt hinter jedem bedingten Sprung zwei Pfeile: einen grünen und einen roten. Der grüne Pfeil zeigt auf den Codeblock der ausgeführt wird, wenn der Sprung ausgeführt wird und der rote den Codeblock, der ausgeführt wird, falls nicht gesprungen wird.

In diesem Modus ist es möglich, Knoten einzuklappen und ihnen auch Namen zu geben („group nodes“). Wir probieren das mit 3 Blöcken aus:

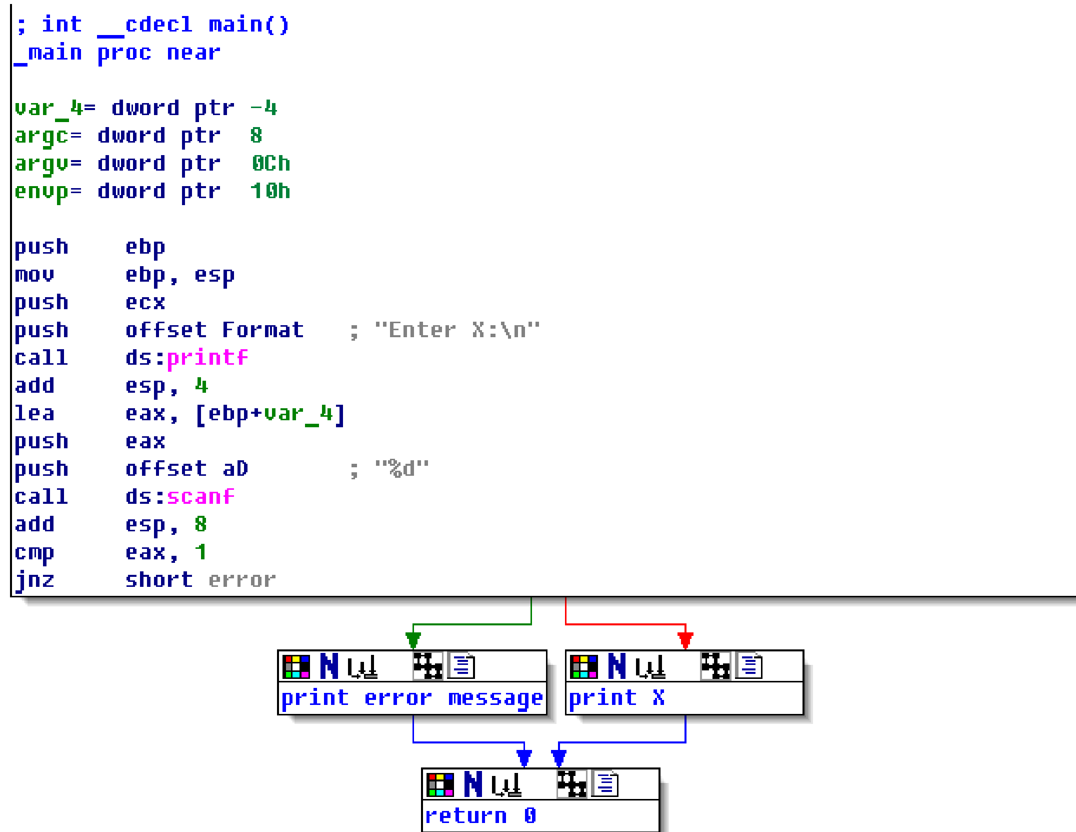


Abbildung 1.19: Graph Modus in IDA mit 3 eingeklappten Knoten

Das ist sehr nützlich. Man kann sagen, dass ein großer Teil der Arbeit eines Reverse Engineers (und eines jeden anderen Forsches) darin besteht, die Menge der zur Verfügung stehenden Informationen zu reduzieren.

MSVC: x86 + OllyDbg

Laden wir unser Programm in OllyDbg und zwingen es dazu zu glauben, dass `scanf()` stets ohne Fehler arbeitet. Wenn die Adresse einer lokalen Variablen an `scanf()` übergeben wird, enthält die Variable zu Beginn einen zufälligen Wert, in diesem Fall `0x6E494714`:

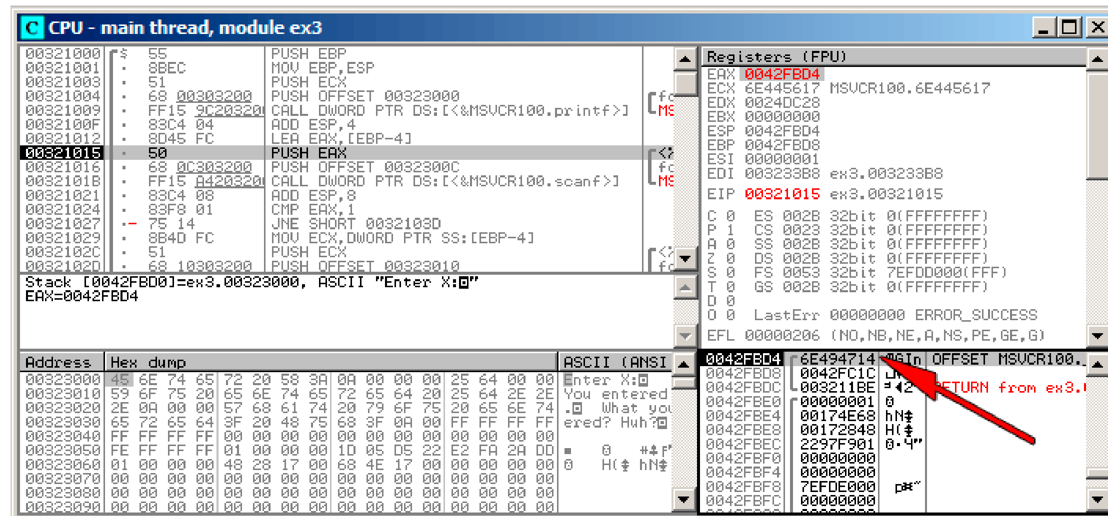


Abbildung 1.20: OllyDbg: Adresse der Variablen an `scanf()` übergeben

Während scanf() ausgeführt wird, geben wir in der Konsole etwas ein, das definitiv keine Zahl ist, z.B. „asdasd“. scanf() beendet sich mit 0 in EAX, was anzeigt, dass ein Fehler aufgetreten ist.

Wir können auch die lokale Variable auf dem Stack überprüfen und stellen fest, dass sie sich nicht verändert hat. Was könnte scanf() hier auch hineinschreiben? Die Funktion hat nichts getan außer 0 zurückzugeben.

Versuchen wir unser Programm zu modifizieren, d.i. zu „hacken“. Rechtsklick auf EAX, in den Optionen finden wir „Set to 1“. Das ist was wir brauchen.

Wir haben jetzt 1 in EAX, sodass die folgende Überprüfung wie gewünscht ausgeführt wird und printf() den Wert der Variablen auf dem Stack ausgibt.

Wenn wir das Programm laufen lassen (F9), sehen wir das Folgende im Konsolenfenster:

Listing 1.62: console window

```
Enter X:
asdasd
You entered 1850296084...
```

Und tatsächlich ist 1850296084 die dezimale Darstellung der Zahl auf dem Stack (0x6E494714)!

MSVC: x86 + Hiew

Unser Programm kann auch als einfaches Beispiel für das Patchen einer Executable dienen. Wir könnten versuchen, die Executable so zu patchen, dass das Programm unabhängig vom Input diesen stets auszugeben.

Angenommen, dass die Executable mit externer `MSVCR*.DLL` (d.h. mit der Option `MD`) kompiliert wurde[73], finden wir die Funktion `main()` am Anfang des `.text` Segments. Öffnen wir die Executable in Hiew und schauen uns den Anfang des `.text` Segments an (Enter, F8, F6, Enter, Enter).

Wir sehen das Folgende:

```
Hiew: ex3.exe
    C:\Polygon\ollydbg\ex3.exe                   FRO --------          a32 PE  .00401000 Hie
.00401000: 55                                   push        ebp
.00401001: 8BEC                                 mov         ebp,esp
.00401003: 51                                   push        ecx
.00401004: 6800304000                           push        000403000 ;'Enter X:' --1
.00401009: FF1594204000                         call        printf
.0040100F: 83C404                               add         esp,4
.00401012: 8D45FC                               lea         eax,[ebp][-4]
.00401015: 50                                   push        eax
.00401016: 680C304000                           push        00040300C --2
.0040101B: FF158C204000                         call        scanf
.00401021: 83C408                               add         esp,8
.00401024: 83F801                               cmp         eax,1
.00401027: 7514                                 jnz        .00040103D --3
.00401029: 8B4DFC                               mov         ecx,[ebp][-4]
.0040102C: 51                                   push        ecx
.0040102D: 6810304000                           push        000403010 ;'You entered %d...'
.00401032: FF1594204000                         call        printf
.00401038: 83C408                               add         esp,8
.0040103B: EB0E                                 jmps       .00040104B --5
.0040103D: 6824304000                          3push        000403024 ;'What you entered?
.00401042: FF1594204000                         call        printf
.00401048: 83C404                               add         esp,4
.0040104B: 33C0                                5xor         eax,eax
.0040104D: 8BE5                                 mov         esp,ebp
.0040104F: 5D                                   pop         ebp
.00401050: C3                                   retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
.00401051: B84D5A0000                           mov         eax,000005A4D ;'  ZM'
1Global 2FilBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 9 1byte 10Leave 11Nak
```

Abbildung 1.21: Hiew: `main()` Funktion

Hiew erkennt ASCIIZ[74] Strings und die Namen importierter Funktionen und zeigt diese an.

[73] dieser Vorgang wird auch „dynamisches Verlinken genannt"

[74] ASCII Zero ( )

Setzen wir den Cursor auf die Adresse .00401027, an der sich der JNZ Befehl, den wir umgehen müssen, befindet, drücken F3 und fügen dann „9090“ (zwei **NOPS!**[75]s) ein.

```
Hiew: ex3.exe
   C:\Polygon\ollydbg\ex3.exe                  □FWO EDITMODE          a32 PE  0000
00000400: 55                                    push        ebp
00000401: 8BEC                                  mov         ebp,esp
00000403: 51                                    push        ecx
00000404: 6800304000                            push        000403000 ;' @0 '
00000409: FF1594204000                          call        d,[000402094]
0000040F: 83C404                                add         esp,4
00000412: 8D45FC                                lea         eax,[ebp][-4]
00000415: 50                                    push        eax
00000416: 680C304000                            push        00040300C ;' @0□'
0000041B: FF158C204000                          call        d,[00040208C]
00000421: 83C408                                add         esp,8
00000424: 83F801                                cmp         eax,1
00000427: 90                                    nop
00000428: 90                                    nop
00000429: 8B4DFC                                mov         ecx,[ebp][-4]
0000042C: 51                                    push        ecx
0000042D: 6810304000                            push        000403010 ;' @0□'
00000432: FF1594204000                          call        d,[000402094]
00000438: 83C408                                add         esp,8
0000043B: EB0E                                  jmps        00000044B
0000043D: 6824304000                            push        000403024 ;' @0$'
00000442: FF1594204000                          call        d,[000402094]
00000448: 83C404                                add         esp,4
0000044B: 33C0                                  xor         eax,eax
0000044D: 8BE5                                  mov         esp,ebp
0000044F: 5D                                    pop         ebp
00000450: C3                                    retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-
1       2NOPs   3       4       5       6       7       8Table  9       10
```

Abbildung 1.22: Hiew: ersetzen von JNZ durch zwei NOPs

Wir drücken F9 (update). Die Executable wird gespeichert und verhält sich wie gewünscht.

Zwei NOPs sind wahrscheinlich nicht der ästhetischte Ansatz. Ein anderer Weg die Executable zu patchen besteht darin, das zweite Byte des Opcodes (den Jump Offset) auf 0 zu setzen, sodass JNZ immer zum nächsten Befehl springt.

Wir könnten auch das Gegenteil tun: das erste Byte durch EB ersetzen und das zweite (Jump Offset) unangetastet lassen. Wir würde einen unbedingten Sprung erhalten,

[75] **NOPS!**

der stets ausgeführt wird. In diesem Fall würde unabhängig vom Input stets die Fehlermeldung ausgegeben.

ARM

ARM: Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.63: Optimierender Keil 6/2013 (Thumb Modus)

```
var_8    = -8

         PUSH    {R3,LR}
         ADR     R0, aEnterX      ; "Enter X:\n"
         BL      __2printf
         MOV     R1, SP
         ADR     R0, aD           ; "%d"
         BL      __0scanf
         CMP     R0, #1
         BEQ     loc_1E
         ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
         BL      __2printf

loc_1A                            ; CODE XREF: main+26
         MOVS    R0, #0
         POP     {R3,PC}

loc_1E                            ; CODE XREF: main+12
         LDR     R1, [SP,#8+var_8]
         ADR     R0, aYouEnteredD___ ; "You entered %d...\n"
         BL      __2printf
         B       loc_1A
```

Die neuen Befehle hier sind CMP und BEQ[76]. CMP verhält sich analog zum x86 Befehl gleichen Namens, er zieht ein Argument vom anderen ab und aktualisiert die Flags, falls nötig.

BEQ springt zu einer anderen Adresse, falls die beiden Operanden gleich waren oder das Ergebnis der letzten Berechnung 0 war oder das Zero Flag auf 1 gesetzt ist. Der Befehl verhält sich wie JZ in x86.

Der Rest ist einfach: der Ausführung verläuft in zwei Zweigen, dann vereinen sich die Zweige an der Stelle wieder, an der 0 als Rückgabewert der Funktion in R0 geschrieben wird, und der Funktionsablauf endet.

ARM64

Listing 1.64: Nicht optimierender GCC 4.9.1 ARM64

```
.LC0:
```

[76](PowerPC, ARM) Branch if Equal

```
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"
.LC3:
        .string "What you entered? Huh?"
f6:
; speichere FP und LR auf dem Stack Frame:
        stp     x29, x30, [sp, -32]!
; setze Stack Frame (FP=SP)
        add     x29, sp, 0
; lade Pointer auf den "Enter X:" String:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
; lade Pointer auf den "%d" String:
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
; berechne Adresse von x auf dem lokalen Stack
        add     x1, x29, 28
        bl      __isoc99_scanf
; scanf() liefert Ergebnis nach W0.
; prüfen:
        cmp     w0, 1
; BNE ist Branch if Not Equal
; also, falls W0<>1, springe zu L2
        bne     .L2
; hier ist W0=1, also kein Fehler
; lade Wert x vom lokalen Stack
        ldr     w1, [x29,28]
; lade Pointer auf den "You entered %d...\n" String:
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
; Code überspringen, der "What you entered? Huh?" ausgibt:
        b       .L3
.L2:
; lade Pointer auf den "What you entered? Huh?" String:
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        bl      puts
.L3:
; return 0
        mov     w0, 0
; wiederherstellen von FP und LR:
        ldp     x29, x30, [sp], 32
        ret
```

Der Kontrollfluss wird in diesem Fall mithilfe von CMP/BNE (Branch if Not Equal) aufgespalten.

Listing 1.65: Optimierender GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0           lui     $gp, 0x42
.text:004006A4           addiu   $sp, -0x28
.text:004006A8           li      $gp, 0x418960
.text:004006AC           sw      $ra, 0x28+var_4($sp)
.text:004006B0           sw      $gp, 0x28+var_18($sp)
.text:004006B4           la      $t9, puts
.text:004006B8           lui     $a0, 0x40
.text:004006BC           jalr    $t9 ; puts
.text:004006C0           la      $a0, aEnterX     # "Enter X:"
.text:004006C4           lw      $gp, 0x28+var_18($sp)
.text:004006C8           lui     $a0, 0x40
.text:004006CC           la      $t9, __isoc99_scanf
.text:004006D0           la      $a0, aD          # "%d"
.text:004006D4           jalr    $t9 ; __isoc99_scanf
.text:004006D8           addiu   $a1, $sp, 0x28+var_10  # branch delay slot
.text:004006DC           li      $v1, 1
.text:004006E0           lw      $gp, 0x28+var_18($sp)
.text:004006E4           beq     $v0, $v1, loc_40070C
.text:004006E8           or      $at, $zero       # branch delay slot, NOP
.text:004006EC           la      $t9, puts
.text:004006F0           lui     $a0, 0x40
.text:004006F4           jalr    $t9 ; puts
.text:004006F8           la      $a0, aWhatYouEntered  # "What you entered?
    Huh?"
.text:004006FC           lw      $ra, 0x28+var_4($sp)
.text:00400700           move    $v0, $zero
.text:00400704           jr      $ra
.text:00400708           addiu   $sp, 0x28

.text:0040070C loc_40070C:
.text:0040070C           la      $t9, printf
.text:00400710           lw      $a1, 0x28+var_10($sp)
.text:00400714           lui     $a0, 0x40
.text:00400718           jalr    $t9 ; printf
.text:0040071C           la      $a0, aYouEnteredD___  # "You entered
    %d...\n"
.text:00400720           lw      $ra, 0x28+var_4($sp)
.text:00400724           move    $v0, $zero
.text:00400728           jr      $ra
.text:0040072C           addiu   $sp, 0x28
```

Übung

Wie wir sehen können, kann der Befehl JNE/JNZ einfach durch JE/JZ eresetzt werden und umgekehrt (oder BNE durch BEQ und umgekehrt). Aber dann müssen die Basisblöcke ebenfalls vertauscht werden. Versuchen Sie dies in einigen der Übungen.

1.9.5 Übung

- http://challenges.re/53

1.10 Zugriff auf übergebene Argumente

Nun haben wir heraus gefunden das die caller Funktion die Argumente zur callee Funtktion über den Stack schiebt. Aber wie greift die callee Funktion auf sie zu?

Listing 1.66: simple example

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

1.10.1 x86

MSVC

Das ist das Ergebnis nach dem kompilieren (MSVC 2010 Express):

Listing 1.67: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8         ; größe = 4
_b$ = 12        ; größe = 4
_c$ = 16        ; größe = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP
```

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; drittes Arguemnt
        push    2 ; zweites Argument
        push    1 ; erstes Argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; return 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Was wir hier sehen ist das die `main()` Funktion drei Zahlen auf den Stack schiebt und `f(int,int,int).` aufruft

Der Argument zugriff innerhalb von `f()` wird organisiert mit der Hilfe von Makros wie zum Beispiel:
`_a$ = 8`, auf die gleiche weise wie Lokale Variablen allerdings mit positiven Offsets (adressiert mit *plus*).

Also adressieren wir die *äussere* Seite des Stack frame indem wir `_a$` Makros zum Wert des EBP Registers addieren

Dann wird der Wert von a in EAX gespeichert. Nachdem die `IMUL` Instruktion ausgeführt wurde, ist der Wert in EAX ein Produkt des Wertes aus EAX und dem Inhalt von `_b`.

Nun addiert `ADD` den Wert in `_c` auf EAX

Der Wert in EAX muss nicht verschoben werden: Der Wert von EAX befindet sich schon wo er sein muss

Beim zurück kehren zur caller Funktion, wird der Wert aus EAX genommen und als Argument für den `printf()` Aufruf benutzt.

MSVC + OllyDbg

Lasst uns die Darstellung in OllyDbg betrachten

Wenn wir die erste Instruktion tracen in `f()` das auf eines der Argumente zugreift (das erste), können wir sehen das EBP auf den Stack Frame zeigt, dieser Frame wird mit dem roten Rechteck markiert dargestellt.

Das erste Element des Stack Frame ist der gespeicherte Wert von EBP, das zweite Element ist RA, das dritte Element ist das erste Funktions Argument, dann folgt das zweite und dritte Funktions Argument.

Um auf das erste Funktions Argument zu zugreifen, muss man lediglich exakt 8 (2 32-Bit Wörter) zu EBP addieren.

OllyDbg erkennt diesen Umstand, und Kommentare zu den entsprechenden Stack Elementen hinzugefügt zum Beispiel:

„RETURN from“ und „Arg1 = …“, etc.

Beachte: Funktions Argumente sind keine Mitglieder des Funktions Stack Frame, sie sind eher Mitglieder des Stack Frame der caller Funktion.

Deswegen, hat OllyDbg die „Arg“ Elemente als Mitglied eines anderen Stackframes identifiziert.

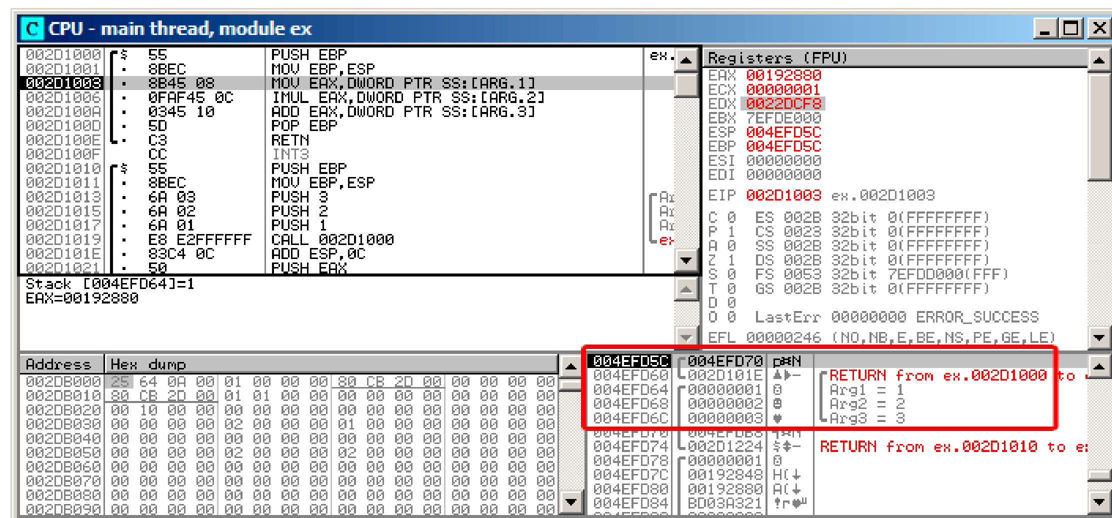


Abbildung 1.23: OllyDbg: inside of f() function

GCC

Lasst uns das gleiche in GCC kompilieren und die Ergebnisse in IDA betrachten:

Listing 1.68: GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8
arg_4   = dword ptr  0Ch
arg_8   = dword ptr  10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; erstes Argument
        imul    eax, [ebp+arg_4] ; zweites Argument
        add     eax, [ebp+arg_8] ; drittes Argument
        pop     ebp
```

```
        retn
f       endp

        public main
main    proc near

var_10  = dword ptr -10h
var_C   = dword ptr -0Ch
var_8   = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     [esp+10h+var_8], 3 ; drittes Argument
        mov     [esp+10h+var_C], 2 ; zweites Argument
        mov     [esp+10h+var_10], 1 ; erstes Argument
        call    f
        mov     edx, offset aD  ; "%d\n"
        mov     [esp+10h+var_C], eax
        mov     [esp+10h+var_10], edx
        call    _printf
        mov     eax, 0
        leave
        retn
main    endp
```

Das Ergebnis ist fast das gleiche aber mit kleineren Unterschieden die wir bereits früher besprochen haben.

Der Stapel-Zeiger wird nicht zurück gesetzt nach den beiden Funktion aufrufen (f und printf), weil die vorletzte LEAVE Instruktion (**??** on page ??) sich um das zurück setzen kümmert.

1.10.2 x64

Die Geschichte bei x86-64 Funktions Argumenten ist ein wenig anders (zumindest für die ersten vier bis sechs) sie werden über die Register übergeben z.b. der callee liest direkt aus den Registern anstatt vom Stack zu lesen.

MSVC

Optimierender MSVC:

Listing 1.69: Optimierender MSVC 2012 x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
```

```
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - erstes Argument
        ; EDX - zweites Argument
        ; R8D - drittes Argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

Wie wir sehen können, die compact Funktion `f()` nimmt alle Argumente aus den Registern.

Die LEA Instruktion wird hier für Addition benutzt, scheinbar hat der Compiler die Instruktion für schneller befunden als die `ADD` Instruktion.

LEA wird auch benutzt in der `main()` Funktion um das erste und das dritte `f()` Argument vor zu bereiten. Der Compiler muss entschieden haben das dies schneller abgearbeitet wird als die Werte in die Register zu laden mit der `MOV` Instruktion.

Lasst uns einen Blick auf nicht optimierte MSVC Ausgabe werfen:

Listing 1.70: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - erstes Argument
                ; EDX - zweites Argument
                ; R8D - drittes Argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; erstes Argument
                mov     edx, 2 ; zweites Argument
```

```
                mov     ecx, 1 ; drittes Argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

Es sieht ein bisschen wie ein Puzzle aus, weil alle drei Argumente aus den Registern auf dem Stack gespeichert werden aus irgend einem Grund.

Dies bezeichnet man als „shadow space“

[77]: So wird sich wahrscheinlich jede Win64 EXE verhalten und alle vier Register Werte auf dem Stack speichern.

Das wird aus zwei Gründen so gemacht:

1) Es ist ziemlich übertrieben ein ganzes Register (oder gar vier Register) zu Reservieren für eine Argument Übergabe, also werden die Argumente über den Stack zugänglich gemacht. 2) Der Debugger weiß immer wo die Funktions Argumente zu finden sind bei einem breakpoint[78].

Also, so können größere Funktionen ihre Eingabe Argumente im „shadow space“ speichern wenn die Funktion auf die Argumente während der Laufzeit zugreifen will, kleinere Funktionen (wie unsere) zeigen dieses Verhalten nicht.

Es liegt in der Verantwortung vom caller den „shadow space“ auf dem Stack zu allozieren.

GCC

Optimierter GCC generiert mehr oder minder verständlichen Code:

Listing 1.71: Optimierender GCC 4.4.6 x64

```
f:
        ; EDI - erstes Argument
        ; ESI - zweites Argument
        ; EDX - drittes Argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret

main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
```

[77] MSDN
[78] MSDN

```
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; Zahl der übergebenen Vector Register
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Nicht optimierender GCC:

Listing 1.72: GCC 4.4.6 x64

```
f:
        ; EDI - erstes Argument
        ; ESI - zweites Argument
        ; EDX - drittes Argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; Zahl der übergebenen Vector Register
        call    printf
        mov     eax, 0
        leave
        ret
```

Bei System V *NIX Systemen ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [79]) ist kein „shadow space“ nötig, aber der callee will vielleicht seine Argumente irgendwo speichern im Fall das keine oder zu wenig Register frei sind.

[79]Auch verfügbar als https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

GCC: uint64_t statt int

Unser Beispiel funktioniert mit 32-Bit *int*, weshalb auch 32-Bit Register Bereiche benutzt werden (mit dem Präfix E-).

Es lassen sich auch ohne Probleme 64-Bit Werte benutzen:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 1.73: Optimierender GCC 4.4.6 x64

```
f       proc near
        imul    rsi, rdi
        lea     rax, [rdx+rsi]
        retn
f       endp

main    proc near
        sub     rsp, 8
        mov     rdx, 3333333344444444h ; drittes Argument
        mov     rsi, 1111111122222222h ; zweites Argument
        mov     rdi, 1122334455667788h ; erstes Argument
        call    f
        mov     edi, offset format ; "%lld\n"
        mov     rsi, rax
        xor     eax, eax ; Anzahl der Vector Register wird übergeben
        call    _printf
        xor     eax, eax
        add     rsp, 8
        retn
main    endp
```

Der Code ist der gleiche, aber diesmal werden die *full size* 64-Bit Register benutzt (mit dem R- Präfix).

1.10.3 ARM

Nicht optimierender Keil 6/2013 (ARM Modus)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
.text:000000B0             main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0        ; "%d\n"
.text:000000D0 E3 18 00 EB      BL      __2printf
.text:000000D4 00 00 A0 E3      MOV     R0, #0
.text:000000D8 10 80 BD E8      LDMFD   SP!, {R4,PC}
```

Die `main()` Funktion ruft einfach zwei weitere Funktionen auf, mit diesen drei werten die dann der ersten Funktion übergeben werden.

Wie bereits angemerkt, auf ARM werden die erste 4 Werte in den ersten vier Registern übergeben (R0-R3).

Die `f()` Funktion, benutzt augenscheinlich die ersten drei Register (R0-R2) als Argumente

Die `MLA` (*Multiplikation Akkumulierung*) Instruktion multipliziert den ersten der beiden Operanden (R3 und R1), addiert den dritten Operanden (R2) zum Produkt und speichert das Ergebnis in das nullte Register (R0), wohin per Standard definiert Funktionen ihre Rückgabe werte speichern.

Multiplikation und Addition in einem (*Fused multiply-add*) ist ist eine sehr nützliche Instruktion. Nebenbei bemerkt gab es eine solche Instruktion auf x86 nicht, bis FMA-Instruktionen in SIMD implementiert wurden. [80].

Die erste Instruktion `MOV R3, R0`, ist anscheinen redundant (es hätte anstatt eine einzelne `MLA` Instruktion benutzt werden können). Der Compiler hat die Instruktion nicht weg optimiert, da das Programm ohne Optimierungen compiliert wurde.

Die BX Instruktion gibt die Kontrolle an die Adresse zurück die im LR Register gespeichert ist. Falls nötig switcht die Instruktion den Prozessor Modus von Thumb zu ARM oder umgekehrt. Das kann nötig sein da die `f()` Funktion nicht weiß von welcher Art Code sie vielleicht aufgerufen wird, ARM oder Thumb. Wenn die Funktion von Thumb Code aufgerufen wird gibt BX nicht nur die Kontrolle an die aufrufende Funktion zurück, sondern switcht auch den Prozessor Modus auf Thumb. Es wird kein switch ausgeführt, falls die Funktion von ARM Code aufgerufen wurde [*ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)A2.3.2]

Optimierender Keil 6/2013 (ARM Modus)

[80] wikipedia

```
.text:00000098             f
.text:00000098 91 20 20 E0       MLA     R0, R1, R0, R2
.text:0000009C 1E FF 2F E1       BX      LR
```

Und hier wurde die `f()` Funktion mit dem Kreil Compiler und allen Optimierungen (/O3) kompiliert .

Die MOV Instruktion wurde weg optimiert und jetzt benutzt MLA alle Eingabe Register und schreibt die Ergebnisse direkt nach R0. Genau da wo die aufrufende Funktion das Ergebnis auslesen und benutzten wird.

Optimierender Keil 6/2013 (Thumb Modus)

```
.text:0000005E 48 43          MULS    R0, R1
.text:00000060 80 18          ADDS    R0, R0, R2
.text:00000062 70 47          BX      LR
```

Die MLA Instruktion ist im Thumb Modus nicht verfügbar, also muss der Compiler den Code für diese beiden Instruktionen (Multiplikation und Addition) separat generieren.

Zu erst multipliziert die MULS Instruktion R0 mit R1 und platziert das Ergebnis im Register R0. Die zweite Instruktion (ADDS) addiert das Ergebnis mit R2 und platziert das Ergebnis wieder im R0 Register.

ARM64

Optimierender GCC (Linaro) 4.9

Hier ist alles ganz einfach. MADD ist einfach eine Instruktion die Multiplikation/Addition verschmelzt (ähnlich wie wir es bei MLA gesehen haben)

Alle drei Argumente werden über den 32-Bit Part des X-Registers übergeben. In der tat, die Argumente sind 32-Bit *int*'s. Das Ergebnis wird in W0 gespeichert.

Listing 1.74: Optimierender GCC (Linaro) 4.9

```
f:
        madd    w0, w0, w1, w2
        ret

main:
; save FP and LR to stack frame:
        stp     x29, x30, [sp, -16]!
        mov     w2, 3
        mov     w1, 2
        add     x29, sp, 0
        mov     w0, 1
        bl      f
        mov     w1, w0
        adrp    x0, .LC7
        add     x0, x0, :lo12:.LC7
```

```
        bl      printf
; return 0
        mov     w0, 0
; restore FP and LR
        ldp     x29, x30, [sp], 16
        ret

.LC7:
        .string "%d\n"
```

Lasst uns nun alle Datentypen nach 64-Bit `uint64_t` konvertieren und testen:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                                0x1111111122222222,
                                0x3333333344444444));
        return 0;
};
```

```
f:
        madd    x0, x0, x1, x2
        ret
main:
        mov     x1, 13396
        adrp    x0, .LC8
        stp     x29, x30, [sp, -16]!
        movk    x1, 0x27d0, lsl 16
        add     x0, x0, :lo12:.LC8
        movk    x1, 0x122, lsl 32
        add     x29, sp, 0
        movk    x1, 0x58be, lsl 48
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret

.LC8:
        .string "%lld\n"
```

Die `f()` Funktion ist die gleiche, nur das jetzt alle 64-Bit X-Register benutzt werden. Lange 64-Bit Werte werden Stückweise in die Register geladen, genauer beschrieben hier: 1.30.3 on page 526.

Nicht optimierender GCC (Linaro) 4.9

Der nicht optimierte Compiler lauf ist redundanter:

```
f:
        sub     sp, sp, #16
        str     w0, [sp,12]
        str     w1, [sp,8]
        str     w2, [sp,4]
        ldr     w1, [sp,12]
        ldr     w0, [sp,8]
        mul     w1, w1, w0
        ldr     w0, [sp,4]
        add     w0, w1, w0
        add     sp, sp, 16
        ret
```

Der Code speichert seine Eingabe Argumente auf dem lokalen Stack, für den Fall das die Funktion die W0..W2 Register benutzen muss das verhindert das überschreiben der Original Argumente, die vielleicht noch in Zukunft gebraucht werden.

Das bezeichnet man auch als *Register Save Area.* [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)][81]. Der Callee, ist hier nicht in der Pflicht die Werte zu speichern. So Ähnlich wie beim „Shadow Space“: 1.10.2 on page 113.

Warum hat der optimierte GCC 4.9 Aufruf dieses Argument weg gelassen? Weil der Compiler in dem Fall zusätzliche Optimierungen gemacht hat. Und erkannt hat das die zusätzlichen Argumente in der weiteren Ausführung des Codes nicht mehr benötigt werden. Und auch das die Register W0..W2 auch nicht weiter benötigt werden.

Wir können auch ein MUL/ADD Instruktions paar sehen anstatt einem einzelnen MADD.

1.10.4 MIPS

Listing 1.75: Optimierender GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000        mult    $a1, $a0
.text:00000004        mflo    $v0
.text:00000008        jr      $ra
.text:0000000C        addu    $v0, $a2, $v0     ; branch delay slot
; Ergebnis lieg in $v0 vor der Rückgabe
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
```

[81] Auch verfügbar als http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
.text:00000010          lui     $gp, (__gnu_local_gp >> 16)
.text:00000014          addiu   $sp, -0x20
.text:00000018          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C          sw      $ra, 0x20+var_4($sp)
.text:00000020          sw      $gp, 0x20+var_10($sp)
; set c:
.text:00000024          li      $a2, 3
; set a:
.text:00000028          li      $a0, 1
.text:0000002C          jal     f
; set b:
.text:00000030          li      $a1, 2            ; branch delay slot
; result in $v0 now
.text:00000034          lw      $gp, 0x20+var_10($sp)
.text:00000038          lui     $a0, ($LC0 >> 16)
.text:0000003C          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040          la      $a0, ($LC0 & 0xFFFF)
.text:00000044          jalr    $t9
; Nimm das Ergebnis der Funktion f() und übergebe
; es als zweites Argument an printf():
.text:00000048          move    $a1, $v0          ; branch delay slot
.text:0000004C          lw      $ra, 0x20+var_4($sp)
.text:00000050          move    $v0, $zero
.text:00000054          jr      $ra
.text:00000058          addiu   $sp, 0x20 ; branch delay slot
```

Die ersten vier Funktoins Argumente werden in vier Register übergeben die das A-Präfix haben.

Es gibt zwei spezial Register in MIPS: HI und LO die das 64-Bit Multiplikation Ergebnis der Ausführung der `MULT` Instruktion enthalten.

Auf diese Register sind nur zugreifbar durch die `MFLO` und die `MFHI` Instruktionen. `MFLO` enthält hier die niedrigen Bits aus dem Multiplikations Ergebnis und speichert diese in $V0. Also wird der höhere Wert des 32-Bit Teils der multiplikation einfach verworfen (der HI Register in halt wird nicht verwendet) . In der Tat: Wir operieren hier auf 32-Bit *int* Daten Typen.

Zum Schluss addiert `ADDU` („Add Unsigned“) den Wert des dritten Argumentes zum Ergebnis.

Es gibt zwei unterschiedliche Addition Instruktionen auf der MIPS Plattform: `ADD` und `ADDU`. Der unterschied zwischen den beiden Instruktionen bezieht sich nicht auf das Vorzeichen (+/-) sondern auf die exceptions. `ADD` kann eine exception werfen bei einem overflow, was manchmal nützlich[82] sein kann und wird auch bei Ada PS[83] unterstützt, zum Beispiel:

`ADDU` wirft keine exception bei einem overflow.

Da C/C++ keine Unterstützung hierfür bietet, sehen wir in unserem Beispiel `ADDU` statt `ADD`.

[82] http://blog.regehr.org/archives/1154
[83] Programmiersprache

Das 32-Bit Ergebnis bleibt übrig in $V0.

In main() existiert nun eine neue Instruktion, die interessant für uns ist: JAL „Jump an Link“).

Der unterschied zwischen JAL und JALR ist das in der ersten Instruktion ein relatives offset hart codiert ist, während JALR zur absoluten Adresse gespeichert in einem Register springt („Jump und Link Register“).

Beide f() und die main() Funktionen liegen innerhalb der gleichen Objekt Datei, also ist die relative Adresse von f() bekannt und fix.

1.11 Mehr zu Rückgabewerten

In x86 wird das Ergebnis einer Funktionsausführung[84] normalerweise über das EAX Register zurückgegeben. Wenn es sich um ein Byte oder einen *char* handelt, dann wird der niedere Teil des Registers EAX (AL) verwendet. Wenn eine Funktion eine *float* Zahl zurückgibt, wird das FPU Register ST(0) stattdessen verwendet.

In ARM wird das Ergebnis üblicherweise über das R0 Register zurückgegeben.

1.11.1 Versuch einen Rückgabewert vom Typ *void* zu verwenden

Was würde passieren, wenn der Rückgabewert der Funktion main() vom Typ *void* und nicht *int* wäre? Der sogenannte Startup-Code ruft main() in etwa wie folgt auf:

```
push envp
push argv
push argc
call main
push eax
call exit
```

Mit anderen Worten:

```
exit(main(argc,argv,envp));
```

Wenn man main() als *void* deklariert muss nichts explizit zurückgegeben werden (mit dem *return* Ausdruck). In diesem Fall wird etwas Zufälliges, das am Ende der Ausführung von main() in EAX steht, das einzige Argument der exit() Funktion. Höchstwahrscheinlich wird es sich um einen Zufallswert handelt, der von der Ausführung der Funktion dort belassen wurde, sodass der Exitcode des Programms pseudozufällig ist.

Wir veranschaulichen diese Tatsache. Beachten Sie, dass die Funktion main() hier den Rückgabetyp *void* hat:

[84] siehe auch: MSDN: Return Values (C++): MSDN

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Kompilieren wir es in Linux.

GCC 4.8.1 hat `printf()` durch `puts()` ersetzt. (wir haben dies vorher gesehen: 1.5.3 on page 28), aber das ist in Ordnung, denn `puts()` liefert die Anzahl der ausgegebenen Zeichen zurück, genau wie `printf()`. Man beachte, dass EAX vor dem Ende von `main()` nicht geleert wird.

Das bedeutet, dass EAX am Ende von `main()` den Wert enthält, den `puts()` dort hinterlassen hat.

Listing 1.76: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Schreiben wir ein Bash Skript, das den Exitcode anzeigt:

Listing 1.77: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

Führen wir es aus:

```
$ tst.sh
Hello, world!
14
```

14 ist Anzahl der ausgegebenen Zeichen. TBT[85]

Wenn wir übrigens C++ in Hex-Rays dekompilieren, stoßen wir häufig auf eine Funktion, die mit dem Destruktor einer Klasse endet:

[85] To be Translated. The presence of this acronym in this place means that the English version has some new/modified content which is to be translated and placed right here.

```
...

call    ??1CString@@QAE@XZ ; CString:: CString(void)
mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs:0, ecx
add     esp, 28h
retn
```

Gemäß C++ Standard gibt der Destruktor nichts zurück, aber wenn Hex-Rays dies nicht erkennt, und davon ausgeht, dass sowohl Destruktor und diese Funktion *int* zurückgeben, finden wir so etwas wie das Folgende im Output:

```
...

        return CString::~CString(&Str);
}
```

1.11.2 Was, wenn wir das Funktionsergebnis nicht verwenden?

`printf()` gibt die Anzahl der erfolgreich ausgegebenen Zeichen zurück, aber das Ergebnis dieser Funktion wird in der Praxis kaum verwendet.

Es ist also möglich eine Funktion aufzurufen, die einen Wert zurückgibt, diesen Wert aber nicht zu verwenden:

```
int f()
{
    // skip first 3 random values:
    rand();
    rand();
    rand();
    // and use 4th:
    return rand();
};
```

Das Ergebnis der Funktion rand() bleibt in allen vier Fällen in EAX.

In den ersten drei Fällen wird der Wert in EAX aber nicht verwendet.

1.11.3 Eine Struktur zurückgeben

Gehen wir nochmals darauf ein, dass der Rückgabewert im EAX Register verbleibt.

Dies ist der Grund dafür, dass alte C Kompiler keine Funktion erzeugen konnten, die einen Wert zurückgeben, der nicht in ein Register passt (normalerweise *int*), aber falls es erforderlich ist, kann man Informationen über Pointer, die der Funktion als Argumente übergeben wurden, zurückgeben.

Wenn also eine Funktion mehrere Werte zurückgeben soll, gibt sie normalerweise nur einen zurück und den Rest über Pointer.

Nun ist es möglich geworden beispielsweise eine komplette Struktur zurückzugeben, aber die Möglichkeit wird nicht besonders häufig genutzt. Wenn eine Funktion eine große Struktur zurückgeben muss, muss der Aufrufer Speicher hierfür reservieren und einen Pointer darauf als erstes Argument übergeben. Dies ist das fast das gleiche wie einen Pointer manuell in das erste Argument zu übergeben, aber der Kompiler verbirgt es.

Ein kleines Beispiel:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

…was wir erhalten, ist (MSVC 2010 /Ox):

```
$T3853 = 8                      ; size = 4
_a$ = 12                        ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC            ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP            ; get_some_values
```

Der Name des Makros um intern einen Pointer auf eine Struktur zu übergeben, ist hier $T3853.

Dieses Beispiel kann mithilfe der C99 Spracherweiterungen umgeschrieben werden:

```
struct s
{
    int a;
```

```
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 1.78: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

Wie wir sehen, füllt die Funktion nur die Felder der Struktur, die durch die aufrufende Funktion angelegt wurden als ob ein Pointer auf die Struktur übergeben worden wäre. Es gibt an dieser Stelle also keine Nachteile bezüglich Performance.

1.12 Pointer

1.12.1 Werte zurückgeben

Pointer werden oft verwendet um Funktionsergebnisse zurückzuliefern (siehe der Fall `scanf()` (1.9 on page 68)).

Zum Beispiel dann, wenn eine Funktion zwei Werte zurückgeben soll.

Beispiel mit globalen Variablen

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;
```

```
void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Dies kompiliert zu:

Listing 1.79: Optimierender MSVC 2010 (/Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H

_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_sum$ = 16              ; size = 4
_product$ = 20          ; size = 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28
        xor     eax, eax
        ret     0
_main   ENDP
```

Schauen wir es uns in OllyDbg an:

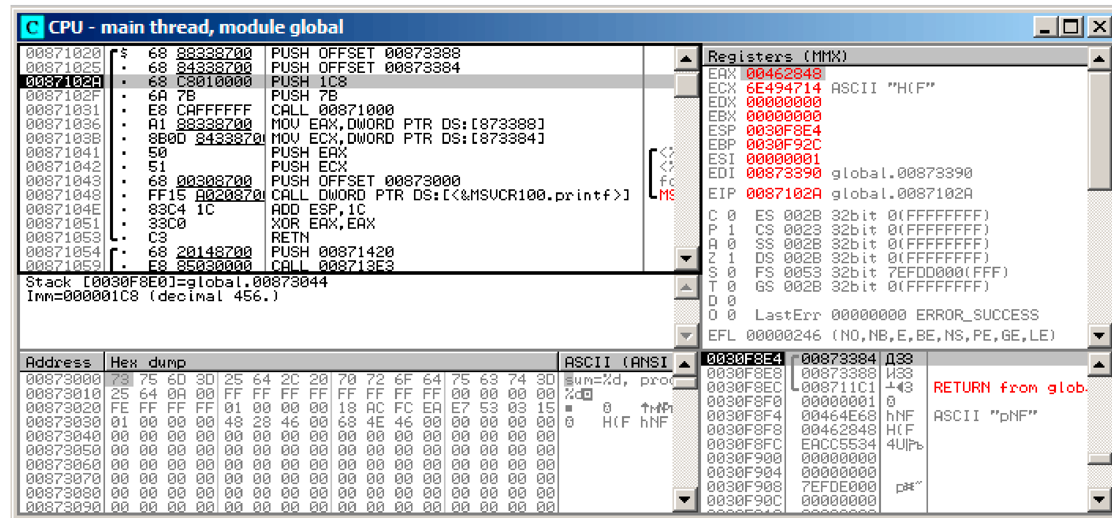


Abbildung 1.24: OllyDbg: Adressen der globalen Variablen werden an `f1()` übergeben

Zuerst werden die Adressen der globalen Variablen an `f1()` übergeben. Wir klicken auf „Follow in dump“ beim Stackelement und wir sehen den Platz, der im Datensegment für die beiden Variablen angelegt wird.

Diese Variablen werden auf Null gesetzt, denn nicht initialisierte Daten (aus BSS) werden gelöscht, bevor die Ausführung beinnt, [siehe *ISO/IEC 9899:TC3 (C C99 standard)*, (2007) 6.7.8p10].

Sie bleiben im Datensegment, was wir durch Drücken von Alt-M und untersuchen der Speicherzuordnung verifizieren können:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00050000	00004000				Map	R	R	
00060000	00001000				Priv	RW	RW	
00070000	00067000				Map	R	R	C:\Windows\System32\loc
00159000	00007000				Priv	RW Gua	RW Gua	
0030D000	00001000				Priv	RW Gua	RW Gua	
0030E000	00002000			Stack of main thread	Priv	RW	RW	
00460000	00005000			Heap	Priv	RW	RW	
004A0000	00007000				Priv	RW	RW	
006B0000	0000C000			Default heap	Priv	RW	RW	
00870000	00001000	global		PE header	Img	R	RWE Cop	
00871000	00001000	global	.text	Code	Img	R E	RWE Cop	
00872000	00001000	global	.rdata	Imports	Img	R	RWE Cop	
00873000	00001000	global	.data	Data	Img	RW	RWE Cop	
00874000	00001000	global	.reloc	Relocations	Img	R	RWE Cop	
6E3E0000	00001000	MSVCR100		PE header	Img	R	RWE Cop	
6E3E1000	000B2000	MSVCR100	.text	Code, imports, exports	Img	R E	RWE Cop	
6E493000	00006000	MSVCR100	.data	Data	Img	RW Cop	RWE Cop	
6E499000	00001000	MSVCR100	.rsrc	Resources	Img	R	RWE Cop	
6E49A000	00005000	MSVCR100	.reloc	Relocations	Img	R	RWE Cop	
755D0000	00001000	Mod_755D		PE header	Img	R	RWE Cop	
755D1000	00003000				Img	R E	RWE Cop	
755D4000	00001000				Img	RW	RWE Cop	
755D5000	00003000				Img	R	RWE Cop	
755E0000	00001000	Mod_755E		PE header	Img	R	RWE Cop	
755E1000	0004D000				Img	R E	RWE Cop	
7562E000	00005000				Img	RW Cop	RWE Cop	
75633000	00009000				Img	R	RWE Cop	

Abbildung 1.25: OllyDbg: Speicherzuordnung

Verfolgen wir den Ablauf (F7) bis zum Start von f1():

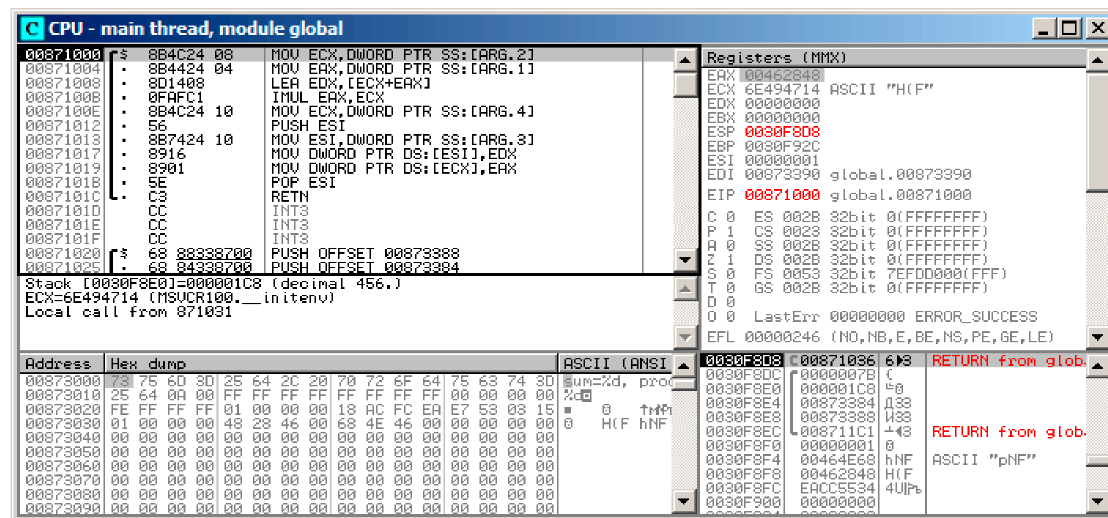


Abbildung 1.26: OllyDbg: f1() beginnt

Zwei Werte sind auf dem Stack sichtbar: 456 (0x1C8) und 123 (0x7B) und außerdem die Adressen der beiden globalen Variablen.

Verfolgen wir den Ablauf bis zum Ende von `f1()`. Im linken unteren Fenster sehen wir wie die Ergebnisse der Berechnung in den globalen Variablen erscheinen:

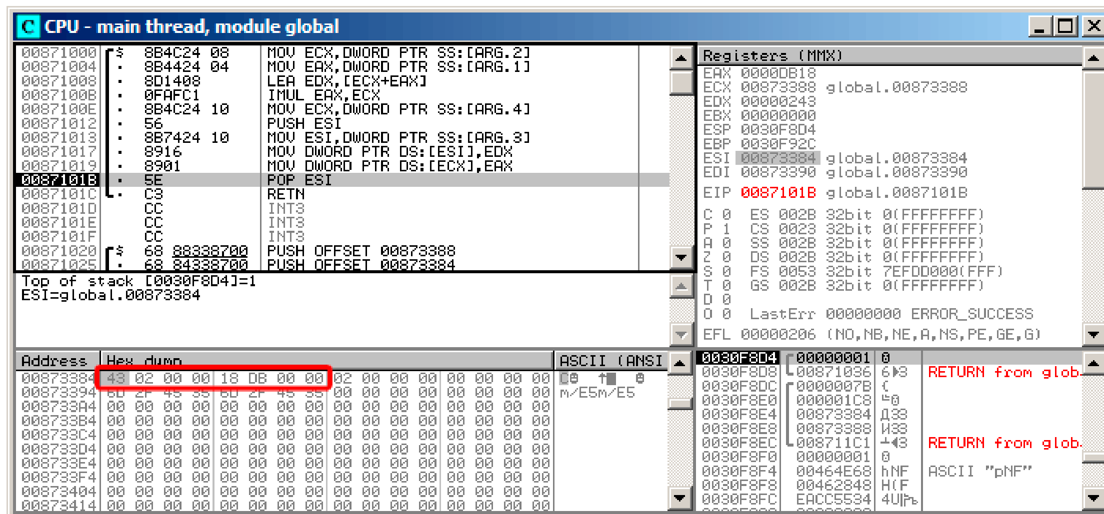


Abbildung 1.27: OllyDbg: Ausführung von `f1()` beendet

Jetzt werden die Werte der globalen Variablen in Register geladen, um dann an `printf()` übergeben zu werden (über den Stack):

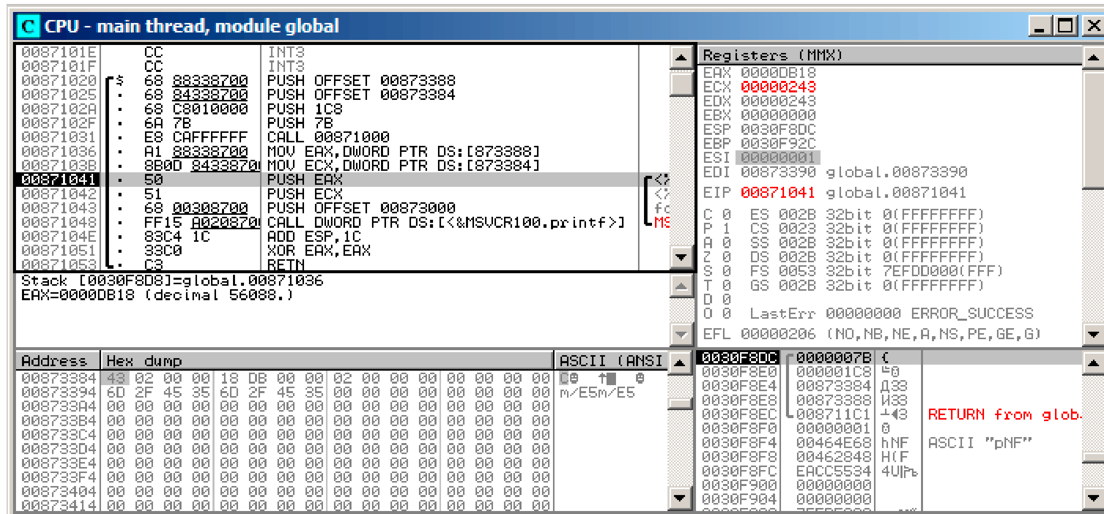


Abbildung 1.28: OllyDbg: Adressen der globalen Variablen werden an `printf()` übergeben

Beispiel mit lokalen Variablen

Verändern wir unser Beispiel ein wenig:

Listing 1.80: `sum` und `product` sind jetzt lokale Variablen

```
void main()
{
        int sum, product; // die beiden sind nun lokale Variablen

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

Der Code von `f1()` wird sich nicht verändern. Nur den Code von `main()` wird sich verändern:

Listing 1.81: Optimierender MSVC 2010 (/Ob0)

```
_product$ = -8          ; size = 4
_sum$ = -4              ; size = 4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
```

```
        push    ecx
        push    456     ; 000001c8H
        push    123     ; 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36
        ret     0
```

Schauen wir es uns erneut mit OllyDbg an. Die Adressen der lokalen Variablen auf dem Stack sind 0x2EF854 und 0x2EF858. Wir erkennen wie diese auf dem Stack abgelegt werden:

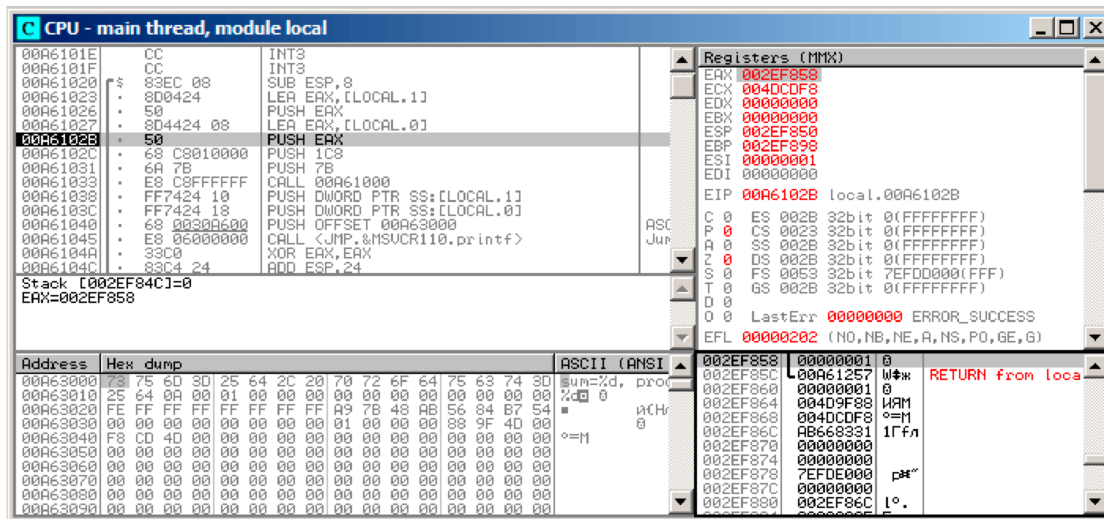


Abbildung 1.29: OllyDbg: Adressen der lokalen Variablen werden auf dem Stack abgelegt

f1() beginnt. Bis hierher befinden sich nur zufällige Werte auf dem Stack an den Adressen 0x2EF854 und 0x2EF858:

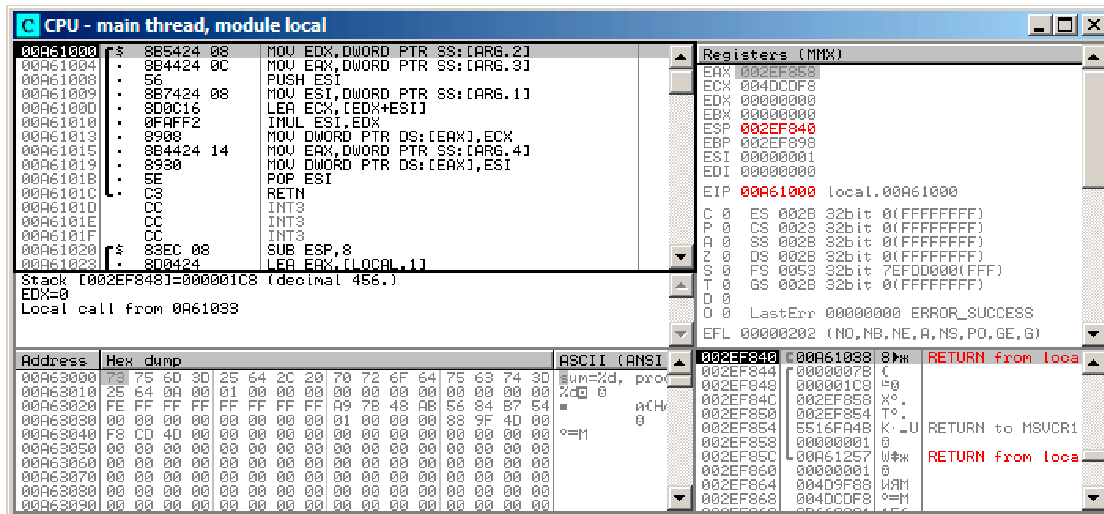


Abbildung 1.30: OllyDbg: f1() beginnt

f1() endet:

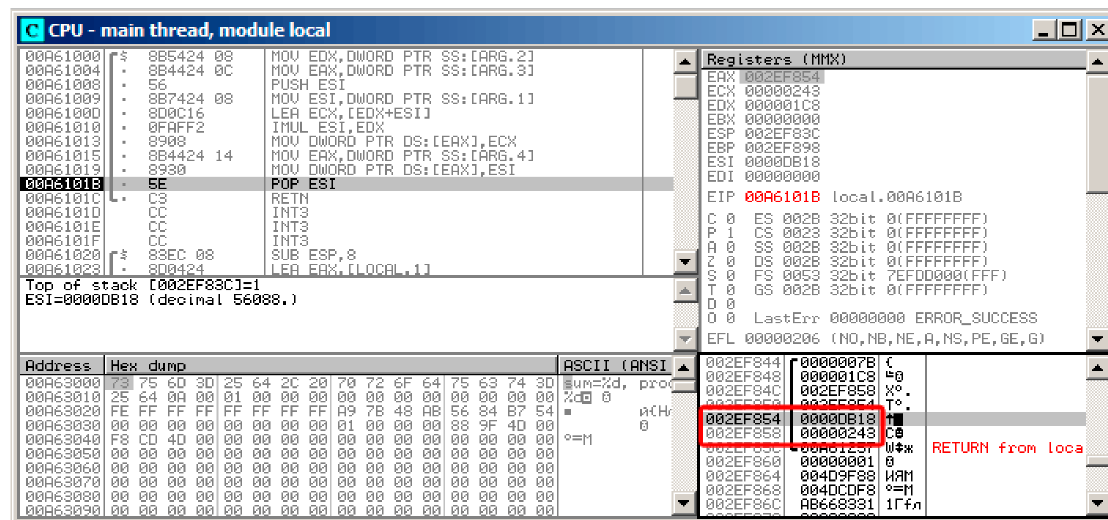


Abbildung 1.31: OllyDbg: Ausführung von f1() endet

Wir finden nun 0xDB18 und 0x243 an den Adressen 0x2EF854 und 0x2EF858. Diese Werte sind die Ergebnisse von f1().

Fazit

f1() kann Pointer auf jede beliebige Speicherstelle zurückgeben. Dies ist die Quintessenz der Nützlichkeit von Pointern.

C++ *references* funktionieren übrigens genau auf die gleiche Weise. Lesen Sie mehr dazu: (**??** on page ??).

1.12.2 Eingabewerte vertauschen

Dies erledigt die Aufgabe für uns:

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
        unsigned char tmp1;
        unsigned char tmp2;

        tmp1=*first;
        tmp2=*second;

        *first=tmp2;
        *second=tmp1;
```

```
};

int main()
{
        // copy string into heap, so we will be able to modify it
        char *s=strdup("string");

        // swap 2nd and 3rd characters
        swap_bytes (s+1, s+2);

        printf ("%s\n", s);
};
```

Wie wir erkennen, werden mit MOVZX Bytes in die niederen 8 Bit von *ECX* und *EBX* geladen (sodass die höherwertigen Teile dieser Register gelöscht werden) und danach werden die Bytes in umgekehrter Reihenfolge zurückgeschrieben.

Listing 1.82: Optimizing GCC 5.4

```
swap_bytes:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+12]
        movzx   ecx, BYTE PTR [edx]
        movzx   ebx, BYTE PTR [eax]
        mov     BYTE PTR [edx], bl
        mov     BYTE PTR [eax], cl
        pop     ebx
        ret
```

Die Adressen der beiden Bytes, die von Argumenten und Ausführung von der Funktion stammen, befinden sich in EDX und EAX.

Wenn wir Pointer verwenden: möglicherweise gibt es keinen besseren Wert diese Aufgabe ohne zu lösen.

1.13 GOTO Operator

Der GOTO Operator wird für gewöhnlich als Anti-Pattern angesehen, vgl. dazu [Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)[86]]. Nichtsdestotrotz kann es er auch sinnvoll eingesetzt werden, siehe [Donald E. Knuth, *Strukt Programming with go to Statements* (1974)[87]] [88].

Hier ist ein sehr einfaches Beispiel:

```
#include <stdio.h>

int main()
{
```

86 http://yurichev.com/mirrors/Dijkstra68.pdf
87 http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf
88 [Dennis Yurichev, *C/C++ programming language notes*] enthält auch einige Beispiele.

```
        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};
```

Dies ehalten wir in MSVC 2012:

Listing 1.83: MSVC 2012

```
$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

Der *Goto* Ausdruck wurde einfach durch einen JMP Befehl ersetzt, der den gleichen Effekt hat: einen unbedingten Sprung an eine andere Stelle. Das zweite `printf()` kann nur durch menschlichen Eingriff ausgeführt werden, z.B. durch einen Debugger oder Patchen des Codes.

Dies könnte auch als einfache Übung zum Patchen von Nutzen sein. Öffnen wir die Executable in Hiew:

```
Hiew: goto.exe
    C:\Polygon\goto.exe                        □FRO --------          a32 PE .00401000
.00401000: 55                                  push        ebp
.00401001: 8BEC                                mov         ebp,esp
.00401003: 6800304000                          push        000403000 ;'begin' --□1
.00401008: FF1590204000                        call        printf
.0040100E: 83C404                              add         esp,4
.00401011: EB0E                                jmps       .000401021 --□2
.00401013: 6808304000                          push        000403008 ;'skip me!' --□3
.00401018: FF1590204000                        call        printf
.0040101E: 83C404                              add         esp,4
.00401021: 6814304000                         2push        000403014 --□4
.00401026: FF1590204000                        call        printf
.0040102C: 83C404                              add         esp,4
.0040102F: 33C0                                xor         eax,eax
.00401031: 5D                                  pop         ebp
.00401032: C3                                  retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

Abbildung 1.32: Hiew

Wir setzen den Cursor auf die Adresse JMP (0x410), drücken F3 (edit), drücken zweimal Null, sodass der Opcode zu EB 00 verändert wird:

```
Hiew: goto.exe
   C:\Polygon\goto.exe                     FWO EDITMODE          a32 PE   00000413
00000400: 55                               push        ebp
00000401: 8BEC                             mov         ebp,esp
00000403: 6800304000                       push        000403000 ;' @0 '
00000408: FF1590204000                     call        d,[000402090]
0000040E: 83C404                           add         esp,4
00000411: EB00                             jmps        000000413
00000413: 6808304000                       push        000403008 ;' @0'
00000418: FF1590204000                     call        d,[000402090]
0000041E: 83C404                           add         esp,4
00000421: 6814304000                       push        000403014 ;' @0'
00000426: FF1590204000                     call        d,[000402090]
0000042C: 83C404                           add         esp,4
0000042F: 33C0                             xor         eax,eax
00000431: 5D                               pop         ebp
00000432: C3                               retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

Abbildung 1.33: Hiew

Das zweite Byte des JMP Opcodes gibt den relativen Offset für den Sprung an; 0 entspricht dabei der Stelle direkt hinter dem aktuellen Befehl.

Auf diese Weise wird JMP den zweiten Aufruf von printf() nicht überspringen.

Wir drücken F9 (save) und verlassen den Editor. Wenn wir die Executable jetzt ausführen, sehen wir dies:

Listing 1.84: Patched executable output

```
C:\...>goto.exe

begin
skip me!
end
```

Das gleiche Ergebnis kann erreicht werden, wenn der JMP Befehl durch 2 NOP Befehle ersetzt wird.

NOP hat einen Opcode von 0x90 und die Länge 1 Byte, sodass wir 2 Befehle als Ersatz für JMP, welcher eine Größe von 2 Byte hat, benötigen.

1.13.1 Dead code

Der zweite Aufruf von printf() wird fachsprachlich auch „dead code“ genannt. Dies bedeutet, dass der Code nie ausgeführt wird. Wenn wir dieses Beispiel mit aktivierter Optimierung kompilieren, entfernt der Compiler den „dead code“ und es gibt keine Spur mehr von ihm:

Listing 1.85: Optimierender MSVC 2012

```
$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main   PROC
        push    OFFSET $SG2981 ; 'begin'
        call    _printf
        push    OFFSET $SG2984 ; 'end'
$exit$4:
        call    _printf
        add     esp, 8
        xor     eax, eax
        ret     0
_main   ENDP
```

Trotzdem hat der Compiler vergessen, den „skip me!“ String ebenfalls zu entfernen.

1.13.2 Übung

Versuchen Sie das gleiche Ergebnis mit ihrem bevorzugten Compiler und Debugger zu erzielen.

1.14 Bedingte Sprünge

1.14.1 einfaches Beispiel

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
```

```
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
};
```

x86

x86 + MSVC

Die Funktions `f_signed()` sieht folgendermaßen aus:

Listing 1.86: Nicht optimierender MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

Der erste Befehl, JLE steht für *Jump if Less or Equal*. Mit anderen Worten, wenn der zweite Operand größer gleich dem ersten ist, wird der Control Flow an die angegebene Adresse bzw. das angegebene Label übergeben. Wenn die Bedingung falsch ist, weil der zweite Operand kleiner ist als der erste, wird der Control Flow nicht verändert und das erste `printf()` wird ausgeführt.

Tmyindexx86!Instruktionen!JNE Der zweite Check ist JNE: *Jump if Not Equal*. Der Control Flow wird nicht verändert, wenn die Operanden gleich sind.

Der dritte Check ist JGE: *Jump if Greater or Equal*—springe, falls der erste Operand größer gleich dem zweiten ist. Wenn also alle drei bedingten Sprünge ausgeführt werden, wird also kein Aufruf von printf() ausgeführt. Dies ist ohne manuellen Eingriff unmöglich. Werfen wir nun einen Blick auf die Funktion f_unsigned(). Diese Funktion macht prinzipiell das gleiche wie f_signed() mit der Ausnahme, dass die Befehle JBE und JAE anstelle von JLE und JGE wie folgt verwendet werden:

Listing 1.87: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
_f_unsigned ENDP
```

Wie bereits erwähnt unterscheiden sich die Verzweigungsbefehle: JBE—*Jump if Below or Equal* und JAE—*Jump if Above or Equal*. Diese Befehle (JA/JAE/JB/JBE) unterscheiden sich von JG/JGE/JL/JLE dadurch, dass sie mit vorzeichenlosen Zahlen arbeiten.

Das ist der Grund warum wir, wenn wir JG/JL anstelle von JA/JB und umgekehrt finden, fast mit Gewissheit sagen können, dass die Variablen vorzeichenbehaftet bzw. vorzeichenlos sind. Hier befindet sich auch die Funktion main(), welche für uns nichts Neues bereithält:

Listing 1.88: main()

```
_main   PROC
        push    ebp
        mov     ebp, esp
```

```
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

x86 + MSVC + OllyDbg

Wir sehen wie die Flags gesetzt werden, indem wir das Beispiel in OllyDbg laufen lassen. Beginnen wir mit `f_unsigned()`; diese Funtion arbeitet mit vorzeichenlosen Zahlen.

CMP wird hier dreimal ausgeführt, aber das die Argumente jeweils identisch sind, sind die Flags jedes Mal die gleichen.

Ergebnis des ersten Vergleichs:

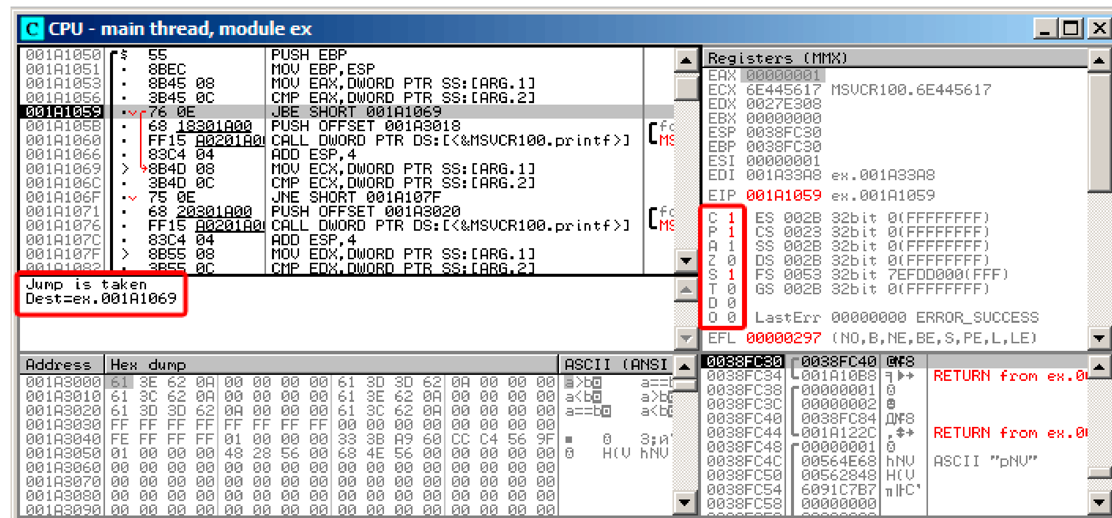


Abbildung 1.34: OllyDbg: `f_unsigned()`: erster bedingter Sprung

Die Flags sind also: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Ihre Namen werden in OllyDbg mit einem Buchstaben abgekürzt.

OllyDbg zeigt an, dass der (JBE) Sprung jetzt ausgeführt werden wird. Und tatsächlich, wenn wir in das Intel-Handbuch (11.1.4 on page 677) schauen, finden wir, dass JBE ausgeführt wird, falls CF=1 oder ZF=1. Da diese Bedingung hier wahr ist, wird der Sprung ausgeführt.

Der nächste bedingte Sprung:

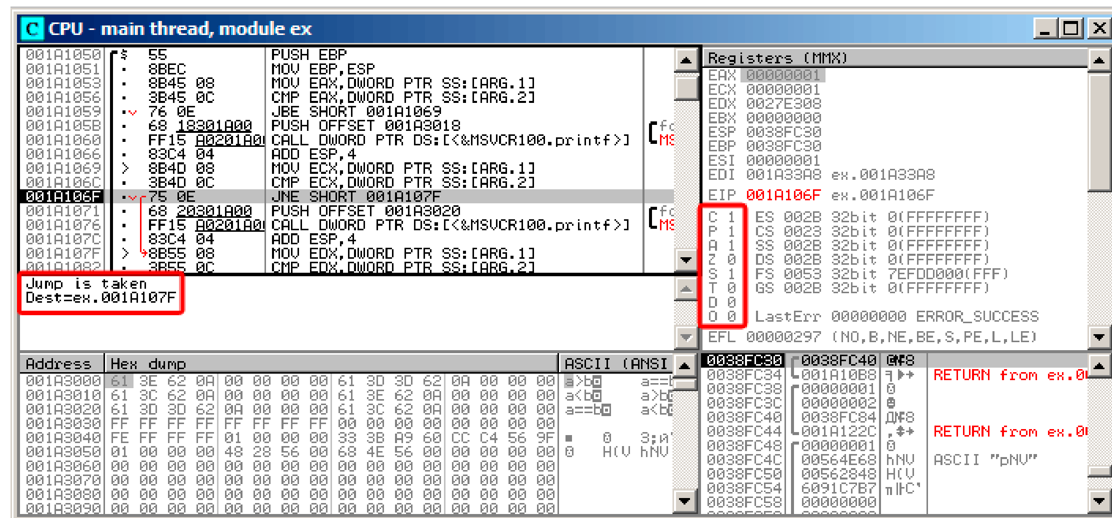


Abbildung 1.35: OllyDbg: `f_unsigned()`: zweiter bedingter Sprung

OllyDbg zeigt an, dass JNZ jetzt ausgeführt werden wird. Tatsächlich wird JNZ ausgeführt, falls ZF=0 (Zero Flag).

Der dritte bedingte Sprung (JNB):

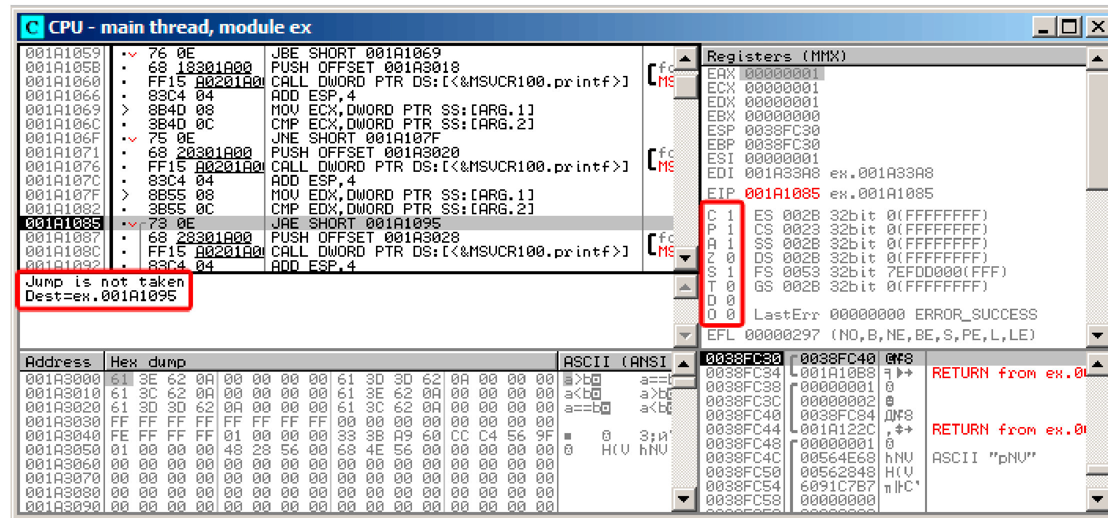


Abbildung 1.36: OllyDbg: f_unsigned(): dritter bedingter Sprung

Im Intel-Handbuch (11.1.4 on page 677) können wir nachlesen, dass JNB ausgeführt wird, falls CF=0 (Carry Flag). Das ist in unserem Beispiel nicht der Falls, sodass das dritte `printf()` ausgeführt wird.

Schauen wir uns nun in OllyDbg die `f_signed()` Funtion an, die mit vorzeichenbehafteten Werte arbeitet. Die Flags werden auf die gleiche Weise gesetzt: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Der erste bedingte Sprung JLE wird danach ausgeführt:

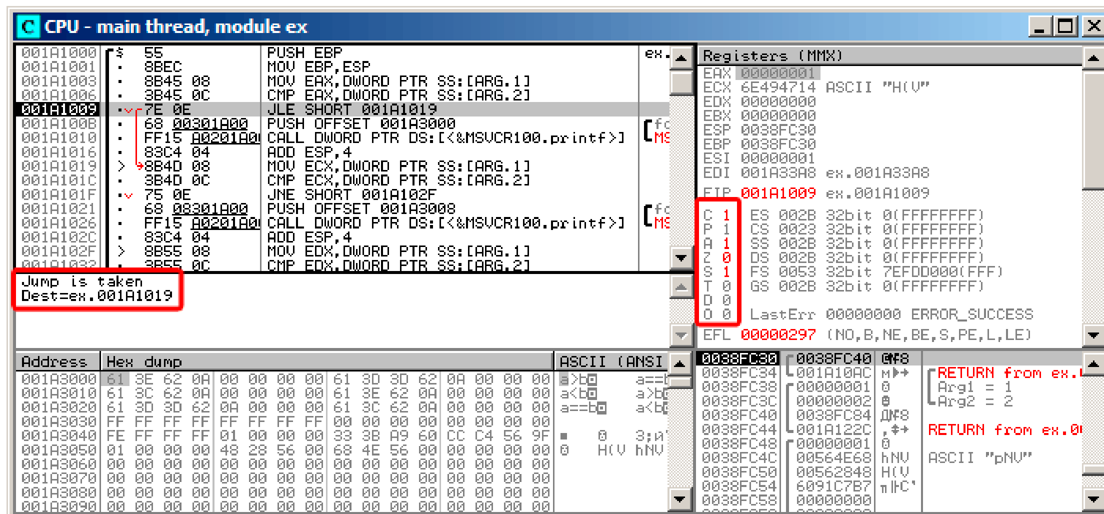


Abbildung 1.37: OllyDbg: `f_signed()`: erster bedingter Sprung

Im Intel-Handbuch (11.1.4 on page 677) können wir nachlesen, dass dieser Befehl ausgeführt wird, falls ZF=1 oder SF≠OF. In userem Fall gilt SF≠OF, sodass der Sprung ausgeführt wird.

Der zweite JNZ bedingte Sprung wird ausgeführt, falls ZF=0 (Zero Flag):

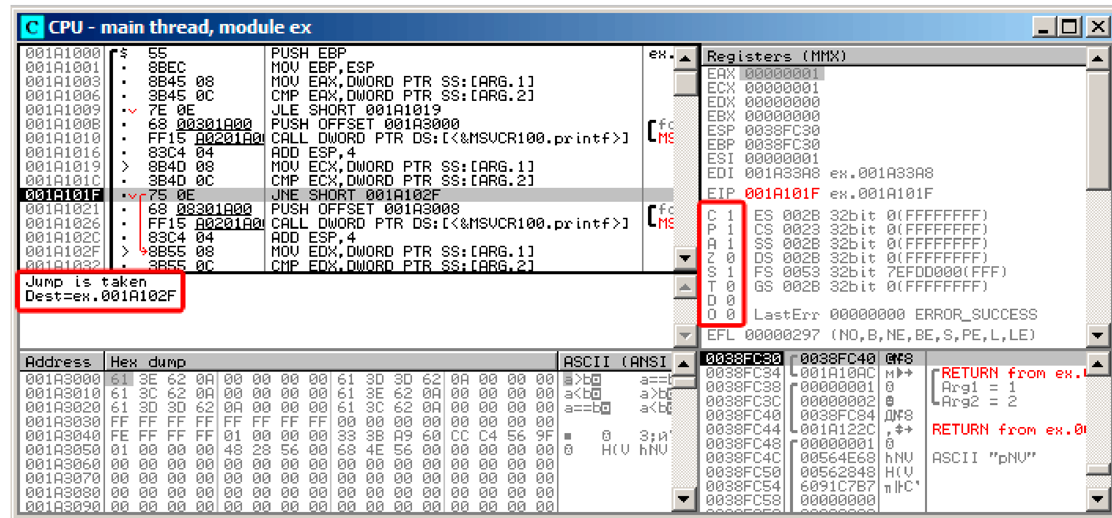


Abbildung 1.38: OllyDbg: f_signed(): zweiter bedingter Sprung

Der dritter bedingte Sprung JBE wird nicht ausgeführt, da dies nur geschieht, falls SF=OF, und dies in unserem Beispiel nicht der Fall ist:

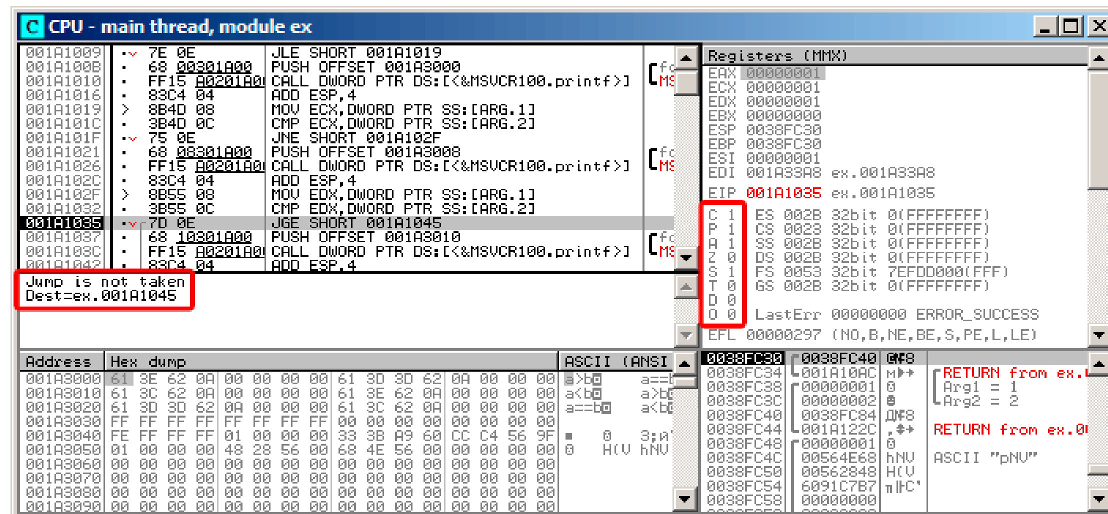


Abbildung 1.39: OllyDbg: f_signed(): dritter bedingter Sprung

x86 + MSVC + Hiew

Wir können versuchen, die Executable so zu verändern, dass die Funktion `f_unsigned()` stets „a==b“ ausgibt, egal was wir eingben. So sieht das ganze in Hiew aus:

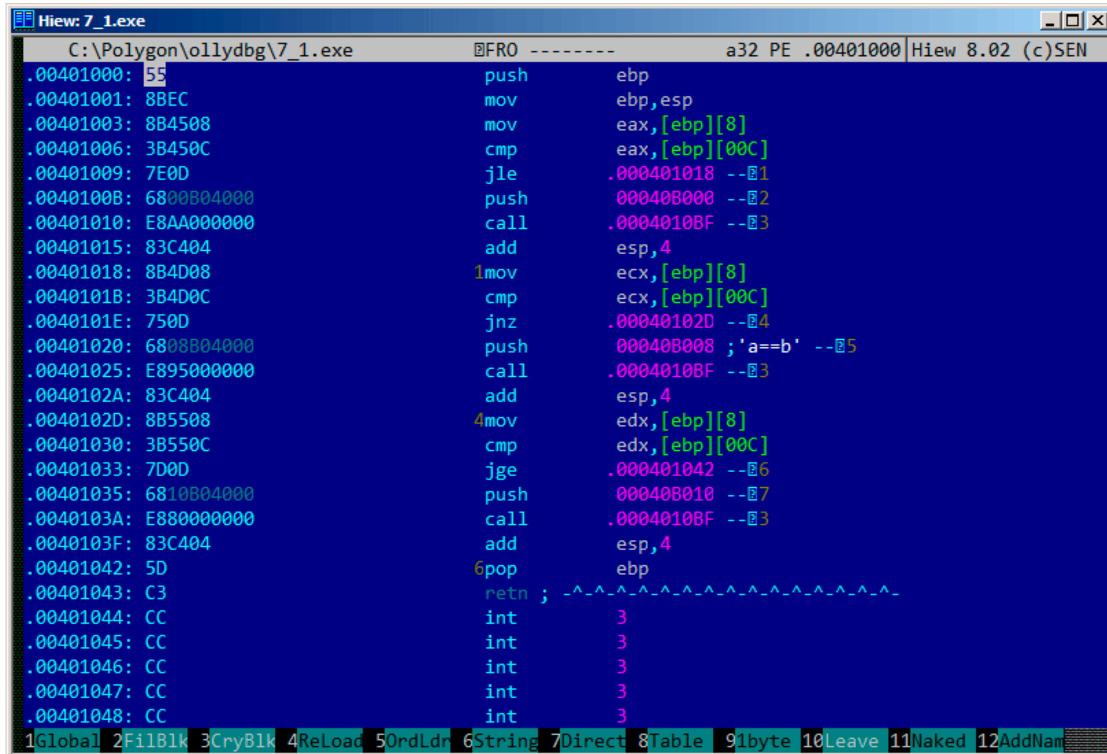


Abbildung 1.40: Hiew: Funktion `f_unsigned()`

Grundsätzlich haben wir drei Dinge zu erzwingen:

- den ersten Sprung stets ausführen;
- den zweiten Sprung nie ausführen;
- den dritten Sprung stets ausführen.

Dadurch können wir den Code Flow so manupulieren, dass das zweite `printf()` immer ausgeführt wird und „a==b“ ausgibt. Drei Befehle (oder Bytes) müssen verändert werden:

- Der erste Sprung wird zu JMP verändert, aber der Jump Offset bleibt gleich.
- Der zweite Sprung könnte manchmal ausgeführt werden, wird aber in jedem Fall zum nächsten Befehl springen, denn wir setzen den Jump Offset auf 0. Bei diesen Befehlen wird der Jump Offset zu der Adresse der nächsten Befehls addiert. Wenn der Offset 0 ist, wird die Ausführung also beim nächsten Befehl fortgesetzt.

- Den dritten Sprung ersetzen wie genau wie den ersten durch `JMP`, damit er stets ausgeführt wird.

Hier ist der veränderte Code:

```
Hiew: 7_1.exe
   C:\Polygon\ollydbg\7_1.exe            FWO EDITMODE        a32 PE  00000434 Hiew 8.02 (c)SEN
00000400: 55                          push        ebp
00000401: 8BEC                        mov         ebp,esp
00000403: 8B4508                      mov         eax,[ebp][8]
00000406: 3B450C                      cmp         eax,[ebp][00C]
00000409: EB0D                        jmps        000000418
0000040B: 6800B04000                  push        00040B000 ;' @ '
00000410: E8AA000000                  call        0000004BF
00000415: 83C404                      add         esp,4
00000418: 8B4D08                      mov         ecx,[ebp][8]
0000041B: 3B4D0C                      cmp         ecx,[ebp][00C]
0000041E: 7500                        jnz         000000420
00000420: 6808B04000                  push        00040B008 ;' @ '
00000425: E895000000                  call        0000004BF
0000042A: 83C404                      add         esp,4
0000042D: 8B5508                      mov         edx,[ebp][8]
00000430: 3B550C                      cmp         edx,[ebp][00C]
00000433: EB0D                        jmps        000000442
00000435: 6810B04000                  push        00040B010 ;' @ '
0000043A: E880000000                  call        0000004BF
0000043F: 83C404                      add         esp,4
00000442: 5D                          pop         ebp
00000443: C3                          retn ; -^-^-^-^-^-^-^-^-^-^-^-^-^-^-^-
00000444: CC                          int         3
00000445: CC                          int         3
00000446: CC                          int         3
00000447: CC                          int         3
00000448: CC                          int         3
1     2NOPs  3     4     5     6     7     8Table  9     10    11    12
```

Abbildung 1.41: Hiew: Veränderte Funktion `f_unsigned()`

Wenn wir es verpassen, einen dieser Sprünge zu verändern, könnten mehrere Aufrufe von `printf()` ausgeführt werden; wir wollen aber nur genau einen Aufruf ausführen.

Nicht optimierender GCC

Nicht optimierender GCC 4.4.1 erzeugt fast identischen Code, aber mit `puts()` (1.5.3 on page 28) anstelle von `printf()`.

Optimierender GCC

Der aufmerksame Leser fragt sich vielleicht, warum CMP mehrmals ausgeführt wird, wenn doch die Flags nach jeder Ausführung dieselben Werte haben.

Vielleicht kann der optimierende MSVC dies nicht leisten, aber der optimierende GCC 4.8.1 gräbt tiefer:

Listing 1.89: GCC 4.8.1 f_signed()

```
f_signed:
```

```
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
        jmp     puts
.L1:
        rep ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

Wir finden auch hier `JMP puts` anstelle von `CALL puts / RETN`.

Dieser Trick wird später erklärt:1.15.1 on page 178.

Diese Sorte x86 Code ist trotzdem selten. MSVC 2012 kann wie es scheint solchen Code nicht erzeugen. Andererseits sind Assemblerprogrammierer sich natürlich der Tatsache bewusst, dass `Jcc` Befehle gestackt werden können. Wenn man solche gestackten Befehle findet, ist es sehr wahrscheinlich, dass der entsprechende Code von Hand geschrieben wurde. Die Funktion `f_unsigned()` ist nicht so ästhetisch:

Listing 1.90: GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx  ; dieser Befehl könnte entfernt werden
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
```

```
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

Trotzdem werden hier immerhin nur zwei statt drei CMP Befehle verwendet.

Die Optimierungsalgorithmen von GCC 4.8.1 sind möglicherweise noch nicht so ausgereift.

ARM

32-bit ARM

Optimierender Keil 6/2013 (ARM Modus)

Listing 1.91: Optimierender Keil 6/2013 (ARM Modus)

```
.text:000000B8                     EXPORT f_signed
.text:000000B8             f_signed              ; CODE XREF: main+C
.text:000000B8 70 40 2D E9         STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1         MOV     R4, R1
.text:000000C0 04 00 50 E1         CMP     R0, R4
.text:000000C4 00 50 A0 E1         MOV     R5, R0
.text:000000C8 1A 0E 8F C2         ADRGT   R0, aAB         ; "a>b\n"
.text:000000CC A1 18 00 CB         BLGT    __2printf
.text:000000D0 04 00 55 E1         CMP     R5, R4
.text:000000D4 67 0F 8F 02         ADREQ   R0, aAB_0       ; "a==b\n"
.text:000000D8 9E 18 00 0B         BLEQ    __2printf
.text:000000DC 04 00 55 E1         CMP     R5, R4
.text:000000E0 70 80 BD A8         LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8         LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2         ADR     R0, aAB_1       ; "a<b\n"
.text:000000EC 99 18 00 EA         B       __2printf
.text:000000EC             ; End of function f_signed
```

Viele Befehle im ARM mode können nur ausgeführt werden, wenn spezielle Flags gesetzt sind. Dies ist beispielsweise oft beim Vergleich von Zahlen der Fall.

Der ADD Befehl zum Beispiel heißt hier intern ADDAL, wobei AL für *Always* (dt. immer) steht, d.h. er wird immer ausgeführt. Die Prädikate werden in den 4 höchstwertigsten Bits des 32-Bit-ARM-Befehls kodiert, dem *condition field*.

Der Befehl B für einen unbedingten Sprung ist tatsächlich doch bedingt und genau wie jeder andere bedingte Sprung kodiert, nut dass er AL im *condition field* hat und dadurch die Flags ignoriert und immer ausgeführt wird.

Der Befehl ADRGT arbeitet wie ADR, wird aber nur ausgeführt, wenn das vorangehende CMP ergeben hat, dass eine der beiden Eingabezahlen größer war als die andere.

Der folgende BLGT Befehl verhält sich genau wie BL und wird nur dann ausgeführt, wenn das Ergebnis des Vergleichs das gleiche war (d.h. größer als). ADRGT schreibt einen Pointer auf den String a>b\n nach R0 und BLGT ruft printf() auf. Das heißt, Befehl mit dem Suffix -GT werden nur ausgeführt, wenn der Wert in R0 (das ist a) größer ist als der Wert in R4 (das ist b).

Im Folgenden finden wir die Befehle ADREQ und BLEQ. Sie verhalten sich wie ADR und BL, werden aber nur ausgeführt, wenn die beiden Operanden des letzten Vergleichs gleich waren. Ein weiteres CMP befindet sich davor (denn die Ausführung von printf() könnte die Flags verändert haben).

Dann finden wir LDMGEFD; dieser Befehl arbeitet genau wie LDMFD[89], wird aber nur ausgeführt, wenn einer der Werte größer gleich dem anderen ist. Der Befehl LDMGEFD SP!, {R4-R6,PC} fungiert als Funktionsepilog, wird aber nur ausgeführt, ewnn $a >= b$ und nur dann wird die Funktionsausführung beendet. Wenn aber diese Bedingung nicht erfüllt ist, d.h. $a < b$, wird der Control Flow zum nächsten „LDMFD SP!, {R4-R6,LR}“ springen, der ebenfalls einen Funktionsepilog darstellt. Dieser Befehl stellt nicht nur den Zustand der R4-R6 Register wieder her, sondern auch LR anstatt **PC!**, dadurch gibt er nichts aus der Funktion zurück. Die letzten beiden Befehle rufen printf() mit dem String «a<b\n» als einzigem Argument auf. Wir haben bereits einen unbedingten Sprung zur Funktion printf() anstelle einer Funktionsrückgabe im Abschnitt «printf() mit mehreren Argumenten» (**??** on page ??) untersucht.

f_unsigned ist ähnlich, nur die Befehle ADRHI, BLHI und LDMCSFD werden hier verwendet. Deren Prädikaten (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) sind analog zu den eben betrachteten, nur eben für vorzeichenlose Werte.

In der Funktion main() finden wir nicht viel Neues:

Listing 1.92: main()

```
.text:00000128                    EXPORT main
.text:00000128               main
.text:00000128 10 40 2D E9        STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3        MOV     R1, #2
.text:00000130 01 00 A0 E3        MOV     R0, #1
.text:00000134 DF FF FF EB        BL      f_signed
.text:00000138 02 10 A0 E3        MOV     R1, #2
.text:0000013C 01 00 A0 E3        MOV     R0, #1
.text:00000140 EA FF FF EB        BL      f_unsigned
.text:00000144 00 00 A0 E3        MOV     R0, #0
.text:00000148 10 80 BD E8        LDMFD   SP!, {R4,PC}
.text:00000148               ; End of function main
```

Auf diese Weise kann man bedingte Sprünge im ARM mode entfernen.

Für eine Begründung warum dies vorteilhaft ist, siehe: **??** on page ??.

[89] LDMFD

In x86 gibt es kein solches Feature, außer dem CMOVcc Befehl, der genau wie MOV funktioniert, aber nur ausgeführt wird, wenn spezielle Flags - normalerweise durch CMP- gesetzt sind.

Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.93: Optimierender Keil 6/2013 (Thumb Modus)

```
.text:00000072             f_signed ; CODE XREF: main+6
.text:00000072 70 B5         PUSH    {R4-R6,LR}
.text:00000074 0C 00         MOVS    R4, R1
.text:00000076 05 00         MOVS    R5, R0
.text:00000078 A0 42         CMP     R0, R4
.text:0000007A 02 DD         BLE     loc_82
.text:0000007C A4 A0         ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8   BL      __2printf
.text:00000082
.text:00000082             loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42         CMP     R5, R4
.text:00000084 02 D1         BNE     loc_8C
.text:00000086 A4 A0         ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8   BL      __2printf
.text:0000008C
.text:0000008C             loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42         CMP     R5, R4
.text:0000008E 02 DA         BGE     locret_96
.text:00000090 A3 A0         ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8   BL      __2printf
.text:00000096
.text:00000096             locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD         POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

Nur der B Befehl im Thumb mode kann mit condition codes versehen werden, sodass der Thumb Code gewöhnlicher aussieht.

BLE ist ein normaler bedingter Sprung *Less than or Equal*, BNE—*Not Equal*, BGE—*Greater than or Equal*.

f_unsigned ist ähnlich, nur dass andere Befehle verwendet werden, wenn mit vorzeichenlosen Werten umgegangen wird: BLS (*Unsigned lower or same*) und BCS (*Carry Set (Greater than or equal)*).

ARM64: Optimierender GCC (Linaro) 4.9

Listing 1.94: f_signed()

```
f_signed:
; W0=a, W1=b
        cmp     w0, w1
        bgt     .L19    ; verzweige, falls größer (a>b)
```

```
        beq     .L20    ; verzweige, falls gleich (a==b)
        bge     .L15    ; verzweige, falls größer gleich (a>=b) (hier nicht
    möglich)
        ; a<b
        adrp    x0, .LC11       ; "a<b"
        add     x0, x0, :lo12:.LC11
        b       puts
.L19:
        adrp    x0, .LC9        ; "a>b"
        add     x0, x0, :lo12:.LC9
        b       puts
.L15:   ; nicht erreichbar
        ret
.L20:
        adrp    x0, .LC10       ; "a==b"
        add     x0, x0, :lo12:.LC10
        b       puts
```

Listing 1.95: f_unsigned()

```
f_unsigned:
        stp     x29, x30, [sp, -48]!
; W0=a, W1=b
        cmp     w0, w1
        add     x29, sp, 0
        str     x19, [sp,16]
        mov     w19, w0
        bhi     .L25    ; verzweige, falls größer (a>b)
        cmp     w19, w1
        beq     .L26    ; verzweige, falls gleich (a==b)
.L23:
        bcc     .L27    ; verzweige, falls Carry Clear (kleiner) (a<b)
; Funktionsepilog, hier nicht erreichbar
        ldr     x19, [sp,16]
        ldp     x29, x30, [sp], 48
        ret
.L27:
        ldr     x19, [sp,16]
        adrp    x0, .LC11       ; "a<b"
        ldp     x29, x30, [sp], 48
        add     x0, x0, :lo12:.LC11
        b       puts
.L25:
        adrp    x0, .LC9        ; "a>b"
        str     x1, [x29,40]
        add     x0, x0, :lo12:.LC9
        bl      puts
        ldr     x1, [x29,40]
        cmp     w19, w1
        bne     .L23    ; verzweige, falls ungleich
.L26:
        ldr     x19, [sp,16]
        adrp    x0, .LC10       ; "a==b"
        ldp     x29, x30, [sp], 48
```

```
        add     x0, x0, :lo12:.LC10
        b       puts
```

Die Kommentare stammen vom Autor. Erstaunlich ist hier, dass der Compiler nicht bemerkt, dass einige Bedingungen unmöglich zu erfüllen sind, sodass Dead Code vorliegt, der nie ausgeführt werden kann.

Übung

Versuchen Sie die Funktionen manuell hinsichtlich Größe und Entfernen redundanter Befehle zu optimieren.

MIPS

Ein wesentliches Feature von MIPS ist das Fehlen von Flags. Der Grund dafür ist offenbar, dass die Analyse von Datenabhängigeiten dadurch vereinfacht wird.

Es gibt Befehle, die Ähnlichkeit mit SETcc in x86 haben:SLT („Set on Less Than“: vorzeichenbehaftete Version) und SLTU (Version ohne Vorzeichen). Diese Befehle setzen das Zielregister auf den Wert 1, falls die Bedingung wahr ist und ansonsten auf 0.

Das Zielregister wird dann mit BEQ („Branch on Equal“) oder BNE („Branch on Not Equal“) überprüft und gegebenenfalls ein Sprung ausgeführt. Dieses Befehlspaar muss also in MIPS für Vergleiche und Verzweigungen verwendet werden. Beginnen wir mit der vorzeichenbehafteten Version unserer Funtion:

Listing 1.96: Nicht optimierender GCC 4.4.5 (IDA)

```
.text:00000000 f_signed:  # CODE XREF: main+18
.text:00000000
.text:00000000 var_10  = -0x10
.text:00000000 var_8   = -8
.text:00000000 var_4   = -4
.text:00000000 arg_0   =  0
.text:00000000 arg_4   =  4
.text:00000000
.text:00000000         addiu   $sp, -0x20
.text:00000004         sw      $ra, 0x20+var_4($sp)
.text:00000008         sw      $fp, 0x20+var_8($sp)
.text:0000000C         move    $fp, $sp
.text:00000010         la      $gp, __gnu_local_gp
.text:00000018         sw      $gp, 0x20+var_10($sp)
; speichere Eingabewerte auf lokalem Stack:
.text:0000001C         sw      $a0, 0x20+arg_0($fp)
.text:00000020         sw      $a1, 0x20+arg_4($fp)
; reload them.
.text:00000024         lw      $v1, 0x20+arg_0($fp)
.text:00000028         lw      $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C         or      $at, $zero ; NOP
; Pseudo-Befehl, entspricht "slt $v0,$v0,$v1"
; setze $v0 auf 1 ,falls $v0<$v1 (b<a) oder ansonsten auf 0:
```

```
.text:00000030         slt     $v0, $v1
; springe zu loc_5c, falls die Bedingung nicht wahr ist
; Pseudo-Befehl, entspricht "beq $v0,$zero,loc_5c":
.text:00000034         beqz    $v0, loc_5C
; gibt "a>b" aus und verlasse
.text:00000038         or      $at, $zero ; branch delay slot, NOP
.text:0000003C         lui     $v0, (unk_230 >> 16) # "a>b"
.text:00000040         addiu   $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000048         or      $at, $zero ; NOP
.text:0000004C         move    $t9, $v0
.text:00000050         jalr    $t9
.text:00000054         or      $at, $zero ; branch delay slot, NOP
.text:00000058         lw      $gp, 0x20+var_10($fp)
.text:0000005C
.text:0000005C loc_5C:                          # CODE XREF: f_signed+34
.text:0000005C         lw      $v1, 0x20+arg_0($fp)
.text:00000060         lw      $v0, 0x20+arg_4($fp)
.text:00000064         or      $at, $zero ; NOP
; prüfe, ob a==b, springe nach loc_90 , falls falsch:
.text:00000068         bne     $v1, $v0, loc_90
.text:0000006C         or      $at, $zero ; branch delay slot, NOP
; Bedingung its wahr, gibt "a==b" aus und verlasse
.text:00000070         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000074         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000078         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000007C         or      $at, $zero ; NOP
.text:00000080         move    $t9, $v0
.text:00000084         jalr    $t9
.text:00000088         or      $at, $zero ; branch delay slot, NOP
.text:0000008C         lw      $gp, 0x20+var_10($fp)
.text:00000090
.text:00000090 loc_90:                          # CODE XREF: f_signed+68
.text:00000090         lw      $v1, 0x20+arg_0($fp)
.text:00000094         lw      $v0, 0x20+arg_4($fp)
.text:00000098         or      $at, $zero ; NOP
; prüfe, ob $v1<$v0 (a<b), setze $v0 auf 1 , falls die Bedingung whar ist.
.text:0000009C         slt     $v0, $v1, $v0
; falls die Bedingung nicht wahr ist (d.h.., $v0==0), springe nach loc_c8:
.text:000000A0         beqz    $v0, loc_C8
.text:000000A4         or      $at, $zero ; branch delay slot, NOP
; Bedingung ist wahr, gib "a<b" aus und verlasse
.text:000000A8         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:000000AC         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:000000B0         lw      $v0, (puts & 0xFFFF)($gp)
.text:000000B4         or      $at, $zero ; NOP
.text:000000B8         move    $t9, $v0
.text:000000BC         jalr    $t9
.text:000000C0         or      $at, $zero ; branch delay slot, NOP
.text:000000C4         lw      $gp, 0x20+var_10($fp)
.text:000000C8
; alle 3 Bedingungen waren falsch, also nur verlassen
.text:000000C8 loc_C8:                                  # CODE XREF:
```

```
    f_signed+A0
.text:000000C8         move    $sp, $fp
.text:000000CC         lw      $ra, 0x20+var_4($sp)
.text:000000D0         lw      $fp, 0x20+var_8($sp)
.text:000000D4         addiu   $sp, 0x20
.text:000000D8         jr      $ra
.text:000000DC         or      $at, $zero ; branch delay slot, NOP
.text:000000DC  # End of function f_signed
```

SLT REG0, REG0, REG1 wird von IDA auf seine kürzere Form reduziert: SLT REG0, REG1. Wir finden dort auch den Pseudo-Befehl BEQZ („Branch if Equal to Zero"), die BEQ REG, $ZERO, LABEL entspricht.

Die vorzeichenlose Version ist identisch, aber SLTU (vorzeichenlose Version, daher das „U" im Namen) wird anstelle von SLT verwendet:

Listing 1.97: Nicht optimierender GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned:  # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10  = -0x10
.text:000000E0 var_8   = -8
.text:000000E0 var_4   = -4
.text:000000E0 arg_0   =  0
.text:000000E0 arg_4   =  4
.text:000000E0
.text:000000E0         addiu   $sp, -0x20
.text:000000E4         sw      $ra, 0x20+var_4($sp)
.text:000000E8         sw      $fp, 0x20+var_8($sp)
.text:000000EC         move    $fp, $sp
.text:000000F0         la      $gp, __gnu_local_gp
.text:000000F8         sw      $gp, 0x20+var_10($sp)
.text:000000FC         sw      $a0, 0x20+arg_0($fp)
.text:00000100         sw      $a1, 0x20+arg_4($fp)
.text:00000104         lw      $v1, 0x20+arg_0($fp)
.text:00000108         lw      $v0, 0x20+arg_4($fp)
.text:0000010C         or      $at, $zero
.text:00000110         sltu    $v0, $v1
.text:00000114         beqz    $v0, loc_13C
.text:00000118         or      $at, $zero
.text:0000011C         lui     $v0, (unk_230 >> 16)
.text:00000120         addiu   $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128         or      $at, $zero
.text:0000012C         move    $t9, $v0
.text:00000130         jalr    $t9
.text:00000134         or      $at, $zero
.text:00000138         lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C:                            # CODE XREF: f_unsigned+34
.text:0000013C         lw      $v1, 0x20+arg_0($fp)
.text:00000140         lw      $v0, 0x20+arg_4($fp)
.text:00000144         or      $at, $zero
.text:00000148         bne     $v1, $v0, loc_170
```

```
.text:0000014C         or      $at, $zero
.text:00000150         lui     $v0, (aAB >> 16)  # "a==b"
.text:00000154         addiu   $a0, $v0, (aAB & 0xFFFF)  # "a==b"
.text:00000158         lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C         or      $at, $zero
.text:00000160         move    $t9, $v0
.text:00000164         jalr    $t9
.text:00000168         or      $at, $zero
.text:0000016C         lw      $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:                          # CODE XREF: f_unsigned+68
.text:00000170         lw      $v1, 0x20+arg_0($fp)
.text:00000174         lw      $v0, 0x20+arg_4($fp)
.text:00000178         or      $at, $zero
.text:0000017C         sltu    $v0, $v1, $v0
.text:00000180         beqz    $v0, loc_1A8
.text:00000184         or      $at, $zero
.text:00000188         lui     $v0, (aAB_0 >> 16)  # "a<b"
.text:0000018C         addiu   $a0, $v0, (aAB_0 & 0xFFFF)  # "a<b"
.text:00000190         lw      $v0, (puts & 0xFFFF)($gp)
.text:00000194         or      $at, $zero
.text:00000198         move    $t9, $v0
.text:0000019C         jalr    $t9
.text:000001A0         or      $at, $zero
.text:000001A4         lw      $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:                          # CODE XREF: f_unsigned+A0
.text:000001A8         move    $sp, $fp
.text:000001AC         lw      $ra, 0x20+var_4($sp)
.text:000001B0         lw      $fp, 0x20+var_8($sp)
.text:000001B4         addiu   $sp, 0x20
.text:000001B8         jr      $ra
.text:000001BC         or      $at, $zero
.text:000001BC  # End of function f_unsigned
```

1.14.2 Betrag berechnen

Eine einfache Funktion:

```
int my_abs (int i)
{
        if (i<0)
                return -i;
        else
                return i;
};
```

Optimierender MSVC

Normalerweise wird folgender Code erzeugt:

Listing 1.98: Optimierender MSVC 2012 x64

```
i$ = 8
my_abs  PROC
; ECX = input
        test    ecx, ecx
; prüfe Vorzeichen des Eingabewertes
; überspringe NEG Befehl, falls Vorzeichen positiv ist
        jns     SHORT $LN2@my_abs
; negiere Wert
        neg     ecx
$LN2@my_abs:
; berechne Ergebnis in EAX:
        mov     eax, ecx
        ret     0
my_abs  ENDP
```

GCC 4.9 macht ungefähr das gleiche.

Optimierender Keil 6/2013: Thumb Modus

Listing 1.99: Optimierender Keil 6/2013: Thumb Modus

```
my_abs PROC
        CMP      r0,#0
; ist Eingabewert größer gleich 0?
; falls ja, überspringe RSBS Befehl
        BGE      |L0.6|
; subtrahiere Eingabewert von 0:
        RSBS     r0,r0,#0
|L0.6|
        BX       lr
        ENDP
```

ARM fehlt ein Befehl zur Negation, sodass der Keil Compiler den „Reverse Subtract“ Befehl verwendet, der mit umgekehrten Operanden subtrahiert.

Optimierender Keil 6/2013: ARM Modus

Es ist im ARM mode möglich, einigen Befehlen condition codes hinzuzufügen und genau das tut der Keil Compiler:

Listing 1.100: Optimierender Keil 6/2013: ARM Modus

```
my_abs PROC
        CMP      r0,#0
; führe "Reverse Subtract" Befehl nur aus,
; falls Eingabewert kleiner als 0 ist:
        RSBLT    r0,r0,#0
        BX       lr
        ENDP
```

Jetzt sind keine bedingten Sprünge mehr übrig und das ist vorteilhaft: **??** on page ??.

Nicht optimierender GCC 4.9 (ARM64)

ARM64 kennt den Befehl NEG zum Negieren:

Listing 1.101: Optimierender GCC 4.9 (ARM64)

```
my_abs:
        sub     sp, sp, #16
        str     w0, [sp,12]
        ldr     w0, [sp,12]
; vergleiche Eingabewert mit dem Inhalt des WZR Registers
; (welches immer 0 enthält)
        cmp     w0, wzr
        bge     .L2
        ldr     w0, [sp,12]
        neg     w0, w0
        b       .L3
.L2:
        ldr     w0, [sp,12]
.L3:
        add     sp, sp, 16
        ret
```

MIPS

Listing 1.102: Optimierender GCC 4.4.5 (IDA)

```
my_abs:
; springe, falls $a0<0:
                bltz    $a0, locret_10
; gib Eingabewert ($a0) in $v0 zurück:
                move    $v0, $a0
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
locret_10:
; negiere Eingabewert und speichere ihn in $v0:
                jr      $ra
; dies ist ein Pseudo-Befehl. Er entspricht "subu $v0,$zero,$a0" ($v0=0-$a0)
                negu    $v0, $a0
```

Hier finden wir einen neuen Befehl: BLTZ („Branch if Less Than Zero“). Es gibt zusätzlich noch den NEGU Pseudo-Befehl, der eine Subtraktion von Null durchführt. Der Suffix „U“ bei SUBU und NEGU zeigt an, dass keine Exception für den Fall eines Integer Overflows geworfen wird.

Verzweigungslose Version?

Man kann auch eine verzweigungslose Version dieses Codes erzeugen. Dies werden wir später betrachten: **??** on page ??.

1.14.3 Ternärer Vergleichsoperator

Der ternäre Vergleichsoperator in C/C++ hat die folgende Syntax:

```
expression ? expression : expression
```

Hier ist ein Beispiel:

```
const char* f (int a)
{
        return a==10 ? "it is ten" : "it is not ten";
};
```

x86

Alte und nicht optimerende Compiler erzeugen Assemblercode als wenn ein `if/else` Ausdruck verwendet wurde:

Listing 1.103: Nicht optimierender MSVC 2008

```
$SG746  DB      'it is ten', 00H
$SG747  DB      'it is not ten', 00H

tv65 = -4 ; wird als temporäre Variable benutzt
_a$ = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        push    ecx
; vergleiche Eingabewert mit 10
        cmp     DWORD PTR _a$[ebp], 10
; springe zu $LN3@f , falls ungleich
        jne     SHORT $LN3@f
; speichere Pointer auf den String in temporärer Variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; springe zum Ende
        jmp     SHORT $LN4@f
$LN3@f:
; speichere Pointer auf den String in temporärer Variable:
        mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; beenden.
; Kopiere Pointer auf den String aus temporärer Variable nach EAX.
        mov     eax, DWORD PTR tv65[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.104: Optimierender MSVC 2008

```
$SG792  DB      'it is ten', 00H
$SG793  DB      'it is not ten', 00H
```

```
_a$ = 8 ; size = 4
_f      PROC
; vergleiche Eingabewert mit 10
        cmp     DWORD PTR _a$[esp-4], 10
        mov     eax, OFFSET $SG792 ; 'it is ten'
; springe zu $LN4@f , falls gleich
        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP
```

Neuere Compiler sind ein wenig präziser:

Listing 1.105: Optimierender MSVC 2012 x64

```
$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f       PROC
; lade Pointer auf beide Strings
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; vergleiche Eingabewert mit 10
        cmp     ecx, 10
; falls gleich, kopiere Wert aus RDX ("it is ten")
; falls nicht, tue nichts. Der Pointer auf den String
; "it is not ten" ist immernoch in RAX.
        cmove   rax, rdx
        ret     0
f       ENDP
```

Optimierender GCC 4.8 für x86 verwendet ebenfalls den `CMOVcc` Befehl, wohingegen der nicht optimierende GCC 4.8 bedingte Sprünge verwendet.

ARM

Optimierender Keil im ARM mode verwendet ebenfalls bedingte Sprungbefehle `ADRcc`:

Listing 1.106: Optimierender Keil 6/2013 (ARM Modus)

```
f PROC
; vergleiche Eingabewert mit 10
        CMP      r0,#0xa
; falls Operanden gleich, kopiere Pointer auf den "it is ten" String nach R0
        ADREQ    r0,|L0.16| ; "it is ten"
; falls Operanden ungleich, kopiere Pointer auf den
; "it is not ten" String nach R0
        ADRNE    r0,|L0.28| ; "it is not ten"
        BX       lr
        ENDP
```

```
|L0.16|
        DCB      "it is ten",0
|L0.28|
        DCB      "it is not ten",0
```

Ohne manuellen Eingriff können die beiden Befehle ADREQ und ADRNE nicht in einem Durchlauf ausgeführt werden.

Optimierender Keil für Thumb mode muss bedingte Sprungbefehle verwenden, da es keine Ladebefehle gibt, die Bedingungsflags unterstützen.

Listing 1.107: Optimierender Keil 6/2013 (Thumb Modus)

```
f PROC
; vergleiche Eingabewert mit 10
        CMP      r0,#0xa
; springe zu |L0.8| , falls gleich
        BEQ      |L0.8|
        ADR      r0,|L0.12| ; "it is not ten"
        BX       lr
|L0.8|
        ADR      r0,|L0.28| ; "it is ten"
        BX       lr
        ENDP

|L0.12|
        DCB      "it is not ten",0
|L0.28|
        DCB      "it is ten",0
```

ARM64

Optimierender GCC (Linaro) 4.9 für ARM64 verwendet auch bedingte Sprünge:

Listing 1.108: Optimierender GCC (Linaro) 4.9

```
f:
        cmp     x0, 10
        beq     .L3             ; verzweige, falls gleich
        adrp    x0, .LC1        ; "it is ten"
        add     x0, x0, :lo12:.LC1
        ret
.L3:
        adrp    x0, .LC0        ; "it is not ten"
        add     x0, x0, :lo12:.LC0
        ret
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
```

Das liegt daran, dass ARM64 über keinen einfachen Ladebefehl mit Bedingungsflags verfügt wie z.B. ADRcc im 32-Bit-ARM-Modus oder CMOVcc in x86.

Es gibt dafür den „Conditional SELect“ Befehl (CSEL)[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5], aber GCC 4.9 scheint nicht ausgereift genug zu sein um ihn in einem solchen Codestück zu verwenden.

MIPS

Leider ist GCC 4.4.5 für MIPS auch nicht besser:

Listing 1.109: Optimierender GCC 4.4.5 (Assemblercode)

```
$LC0:
        .ascii  "it is not ten\000"
$LC1:
        .ascii  "it is ten\000"
f:
        li      $2,10                   # 0xa
; vergleiche $a0 und 10, springe, falls gleich:
        beq     $4,$2,$L2
        nop ; branch delay slot

; lasse Adresse des "it is not ten" Strings in $v0 und beende:
        lui     $2,%hi($LC0)
        j       $31
        addiu   $2,$2,%lo($LC0)

$L2:
; lasse Adresse des "it is ten" Strings in $v0 und beende:
        lui     $2,%hi($LC1)
        j       $31
        addiu   $2,$2,%lo($LC1)
```

Schreiben wir es mit `if/else`

```
const char* f (int a)
{
        if (a==10)
                return "it is ten";
        else
                return "it is not ten";
};
```

Interessanterweise war der optimierende GCC 4.8 für x86 ebenfalls in der Lage `CMOVcc` hier zu verwenden:

Listing 1.110: Optimierender GCC 4.8

```
.LC0:
        .string "it is ten"
.LC1:
        .string "it is not ten"
f:
```

```
.LFB0:
; vergleiche Eingabewert mit 10
        cmp     DWORD PTR [esp+4], 10
        mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
        mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; wenn Operanden ungleich, kopiere EDX nach EAX
; falls nich, tue nichts
        cmovne  eax, edx
        ret
```

Optimierender Keil im ARM mode erzeugt identischen Code zu Listing.1.106. Der optimierende MSVC 2012 ist hingegen (noch) nicht so gut.

Fazit

Warum versuchen optimierende Compiler bedingte Sprünge zu entfernen? Mehr dazu finden Sie hier: **??** on page ??.

1.14.4 Minimale und maximale Werte berechnen

32-bit

```
int my_max(int a, int b)
{
        if (a>b)
                return a;
        else
                return b;
};

int my_min(int a, int b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Listing 1.111: Nicht optimierender MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; vergleiche A und B:
        cmp     eax, DWORD PTR _b$[ebp]
; springe, falls A größer gleich B:
        jge     SHORT $LN2@my_min
; lade A ansonsten erneut nach EAX und springe zum Ende
        mov     eax, DWORD PTR _a$[ebp]
```

```
        jmp     SHORT $LN3@my_min
        jmp     SHORT $LN3@my_min ; redundantes JMP
$LN2@my_min:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
        pop     ebp
        ret     0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
; vergleiche A und B:
        cmp     eax, DWORD PTR _b$[ebp]
; springe, falls A kleiner gleich B:
        jle     SHORT $LN2@my_max
; lade A ansonsten erneut nach EAX und springe zum Ende
        mov     eax, DWORD PTR _a$[ebp]
        jmp     SHORT $LN3@my_max
        jmp     SHORT $LN3@my_max ; this is redundant JMP
$LN2@my_max:
; return B
        mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
        pop     ebp
        ret     0
_my_max ENDP
```

Diese beiden Funktionen unterscheiden sich nur hinsichtliche der bedingten Sprungbefehle: JGE („Jump if Greater or Equal“) wird in der ersten verwendet und JLE („Jump if Less or Equal“) in der zweiten.

Hier gibt es jeweils einen unnötigen JMP Befehl pro Funtion, den MSVC wahrscheinlich fehlerhafterweise dort belassen hat.

Verzweigungslos

ARM im Thumb mode erinnert uns an den x86 Code:

Listing 1.112: Optimierender Keil 6/2013 (Thumb Modus)

```
my_max PROC
; R0=A
; R1=B
; vergleiche A und B:
        CMP      r0,r1
; verzweige, falls A größer B:
        BGT      |L0.6|
; ansonsten (A<=B) liefere R1 (B) zurück:
```

```
        MOVS     r0,r1
|L0.6|
; return
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; vergleiche A und B:
        CMP      r0,r1
; verzweige, falls A kleiner B:
        BLT      |L0.14|
; ansonsten (A>=B) liefere R1 (B) zurück:
        MOVS     r0,r1
|L0.14|
; Rückgabe
        BX       lr
        ENDP
```

Die Funktionen unterscheiden sich in den Verzweigebefehlen: BGT und BLT. Es ist möglich im ARM mode conditional codes zu verwenden, sodass der Code kürzer ist.

MOVcc wird nur ausgeführt, wenn die Bedingung erfüllt (d.h. wahr) ist:

Listing 1.113: Optimierender Keil 6/2013 (ARM Modus)

```
my_max PROC
; R0=A
; R1=B
; vergleiche A und B:
        CMP      r0,r1
; gib B anstatt A zurück, indem B nach R0 geschrieben wird
; dieser Befehl wird nur ausgeführt, falls A<=B (deshalb, LE - Less or Equal)
; wenn der Befehl nicht ausgeführt wird (im Falle von A>B),
; ist A immer noch im R0 Register
        MOVLE    r0,r1
        BX       lr
        ENDP

my_min PROC
; R0=A
; R1=B
; vergleiche A und B:
        CMP      r0,r1
; gib B anstatt A zurück, indem B nach R0 geschrieben wird
; dieser Befehl wird nur ausgeführt, falls A>=B (deshalb, GE - Greater or
    Equal)
; wenn der Befehl nicht ausgeführt wird (im Falle von A<B),
; ist A immer noch im R0 Register
        MOVGE    r0,r1
        BX       lr
        ENDP
```

Optimierender GCC 4.8.1 und der optimierende MSVC 2013 können den CMOVcc Befehl verwenden, der analog zu MOVcc in ARM funktioniert:

Listing 1.114: Optimierender MSVC 2013

```
my_max:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; vergleiche A und B:
        cmp     edx, eax
; falls A>=B, lade Wert A nach EAX
; ansonsten (falls A<B) führe Befehl ohne Auswirkung aus
        cmovge  eax, edx
        ret

my_min:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; vergleiche A und B:
        cmp     edx, eax
; falls A<=B, lade Wert A nach EAX
; ansonsten (falls A>B) führe Befehl ohne Auswirkung aus
        cmovle  eax, edx
        ret
```

64-bit

```
#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
        if (a>b)
                return a;
        else
                return b;
};

int64_t my_min(int64_t a, int64_t b)
{
        if (a<b)
                return a;
        else
                return b;
};
```

Hier findet ein unnötiges Verschieben von Variablen statt, aber der Code ist verständlich:

Listing 1.115: Nicht optimierender GCC 4.9.1 ARM64

```
my_max:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        ble     .L2
        ldr     x0, [sp,8]
        b       .L3
.L2:
        ldr     x0, [sp]
.L3:
        add     sp, sp, 16
        ret

my_min:
        sub     sp, sp, #16
        str     x0, [sp,8]
        str     x1, [sp]
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        cmp     x1, x0
        bge     .L5
        ldr     x0, [sp,8]
        b       .L6
.L5:
        ldr     x0, [sp]
.L6:
        add     sp, sp, 16
        ret
```

Verzweigungslos
Die Funtionsargumente müssen nicht vom Stack geladen werden, da sie sich bereits in den Registern befinden:

Listing 1.116: Optimierender GCC 4.9.1 x64

```
my_max:
; X0=A
; X1=B
; vergleiche A und B:
        cmp     x0, x1
; setze X0 (A) auf X0, falls X0>=X1 oder A>=B (größer gleich)
; setze X1 (B) auf X0, falls A<B
        csel    x0, x0, x1, ge
        ret

my_min:
; X0=A
; X1=B
```

```
; vergleiche A und B:
        cmp     x0, x1
; setze X0 (A) auf X0, falls X0<=X1 oder A<=B (kleiner gleich)
; setze X1 (B) auf X0, falls A>B
        csel    x0, x0, x1, le
        ret
```

MSVC 2013 tut beinahe das gleiche:

ARM64 verfügt über den CSEL Befehl, der genau wie MOVcc in ARM oder CMOVcc in x86 arbeitet; er hat lediglich einen anderen Namen: „Conditional SELect".

Listing 1.117: Optimierender GCC 4.9.1 ARM64

```
my_max:
; X0=A
; X1=B
; vergleiche A und B:
        cmp     x0, x1
; setze X0 (A) auf X0, falls X0>=X1 oder A>=B (größer gleich)
; setze X1 (B) auf X0, falls A<B
        csel    x0, x0, x1, ge
        ret

my_min:
; X0=A
; X1=B
; vergleiche A und B:
        cmp     x0, x1
; setze X0 (A) auf X0, falls X0<=X1 oder A<=B (kleiner gleich)
; setze X1 (B) auf X0, falls A>B
        csel    x0, x0, x1, le
        ret
```

MIPS

Leider ist GCC 4.4.5 für MIPS nicht so gut:

Listing 1.118: Optimierender GCC 4.4.5 (IDA)

```
my_max:
; setze $v1 auf 1 ,falls $a1<$a0, ansonsten (falls $a1>$a0) lösche:
                slt     $v1, $a1, $a0
; springe, falls $v1 ist 0 (oder $a1>$a0):
                beqz    $v1, locret_10
; dies ist der branch delay slot
; bereite $a1 in $v0 vor, falls verzweigt wird
                move    $v0, $a1
; wird nicht verzweigt, bereite $a0 in $v0 vor:
                move    $v0, $a0

locret_10:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

```
; die min() Function ist identisch, aber die Eingabeoperanden
; des SLT Befehls sind vertauscht.
my_min:
                slt     $v1, $a0, $a1
                beqz    $v1, locret_28
                move    $v0, $a1
                move    $v0, $a0

locret_28:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

Vergessen Sie nicht die *branch delay slots*: der erste MOVE wird *vor* BEQZ ausgeführt, der zweite MOVE wird nur dann ausgeführt, wenn die Verzweigung nicht genommen wird.

1.14.5 Fazit

x86

Hier ist der grobe Aufbau eines bedingten Sprungs:

Listing 1.119: x86

```
CMP register, register/value
Jcc true ; cc=condition code
false:
;... dieser Code wird ausgeführt, wenn der Vergleich falsch ergibt ...
JMP exit
true:
;... dieser Code wird ausgeführt, wenn der Vergleich wahr ergibt ...
exit:
```

ARM

Listing 1.120: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
;... dieser Code wird ausgeführt, wenn der Vergleich falsch ergibt ...
JMP exit
true:
;... dieser Code wird ausgeführt, wenn der Vergleich wahr ergibt ...
exit:
```

MIPS

Listing 1.121: prüfe auf Null

```
BEQZ REG, label
...
```

Listing 1.122: Prüfe auf kleiner Null

```
BLTZ REG, label
...
```

Listing 1.123: Prüfe auf Gleichheit

```
BEQ REG1, REG2, label
...
```

Listing 1.124: Prüfe auf Ungleichheit

```
BNE REG1, REG2, label
...
```

Listing 1.125: Prüfe auf größer, größer als (vorzeichenbehaftet)

```
BEQ REG1, label
...
```

Listing 1.126: Prüfe auf kleiner, kleiner als (vorzeichenlos)

```
BEQ REG1, label
...
```

Ohne Verzweigung

Wenn der Rumpf eines bedingten Ausdrucks sehr kurz ist, kann der bedingten Move-Befehl verwendet werden: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

ARM

Es ist im ARM mode möglich, Bedingungssuffixe (engl. condition code) für manchen Befehle zu verwenden:

Listing 1.127: ARM (ARM Modus)

```
CMP register, register/value
instr1_cc ; dieser Befehl wird ausgeführt, wenn der condition code falsch
    ergibt
instr2_cc ; dieser Befehl wird ausgeführt, wenn der condition code wahr
    ergibt
...
;etc...
```

Natürlich gibt es keine Obergrenze für die Anzahl an Befehlen mit conditional codes, solange diese die CPU Flags nicht verändern.

Im Thumb mode gibt es den IT Befehl, der es erlaubt, zusätzliche Suffixe an die vier folgenden Befehle anzuhängen. Mehr dazu unter: 1.19.7 on page 302.

Listing 1.128: ARM (Thumb Modus)

```
CMP register, register/value
ITEEE EQ ; setze folgende Suffixe: if-then-else-else-else
instr1   ; Befehl wird ausgeführt, wenn Bedingung wahr ist
instr2   ; Befehl wird ausgeführt, wenn Bedingung falsch ist
instr3   ; Befehl wird ausgeführt, wenn Bedingung falsch ist
instr4   ; Befehl wird ausgeführt, wenn Bedingung falsch ist
```

1.14.6 Übung

(ARM64) Versuchen Sie den Code in Listing.1.108 so neu zu schreiben, dass alle Befehle mit bedingten Sprüngen durch den CSEL Befehl ersetzt werden.

1.15 switch()/case/default

1.15.1 Kleine Anzahl von Fällen

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

Nicht optimierender MSVC

Ergebnis (MSVC 2010):

Listing 1.129: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
```

```
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
    je     SHORT $LN4@f
    cmp    DWORD PTR tv64[ebp], 1
    je     SHORT $LN3@f
    cmp    DWORD PTR tv64[ebp], 2
    je     SHORT $LN2@f
    jmp    SHORT $LN1@f
$LN4@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN3@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN2@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN7@f
$LN1@f:
    push   OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN7@f:
    mov    esp, ebp
    pop    ebp
    ret    0
_f    ENDP
```

Unsere Funktionen mit ein paar Fällen in switch() ist analog zu dieser Konstruktion:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

Wenn wir mit einem switch() mit einigen wenigen Fällen arbeiten, ist es unmöglich sicher zu sein, dass es sich tatsächlich im Quellcode um ein switch() handelt und nicht um eine Reihe von if()-Anweisungen.

Das bedeutet, dass switch() ein syntaktischer Zucker für eine große Anzahl von verschachtelten if()-Anweisungen ist.

Hier ist nichts Neues für uns im erzeugten Code, mit der Ausnahme, dass der Compiler die Eingabevariable a in einer temporären Variable `tv64` speichert[90].

Wenn wir diesen Code mit GCC 4.4.1 kompilieren, erhalten wir fast das gleiche Ergebnis, sogar unter Verwendung maximaler Optimierung (Option -03).

Optimierender MSVC

Aktivieren wir nun Optimierung in MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

Listing 1.130: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH,
    00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp    _printf
_f    ENDP
```

Hier finden wir ein paar schmutzige Tricks.

Zunächst: der Wert von a wird in EAX abgelegt und 0 wird davon abgezogen. Klingt absurd, muss aber getan werden, um zu prüfen, ob der Wert in EAX null ist. Falls ja, wird das ZF Flag gesetzt (denn 0 minus 0 ist 0) und der erste bedingte Sprung JE (*Jump if Equal*) oder synonym JZ —*Jump if Zero*) wird ausgeführt und der Control Flow wird an das Label $LN4@f übergeben, an dem die Nachricht `'zero'` ausgegeben wird. Wenn der erste Sprung nicht ausgeführt wird, wird 1 vom Eingabewert abgezogen und wenn an irgendeinem Punkt der Ausführung ein Ergebnis von 0 auftritt, wird der zugehörige Sprung ausgeführt.

Falls aber keiner der Sprünge ausgeführt wird, wird der Control Flow mit dem String Argument `'something unknown'` an `printf()` übergeben.

[90] Lokale Variablen im Stack haben den Präfix tv—so benennt MSVC interne Variablen für seine Zwecke

Zweitens: wir sehen etwas für uns Ungewohntes: ein Pointer auf einen String wird in der Variablen a abgelegt und anschließend wird `printf()` nicht über `CALL`, sondern per `JMP` aufgerufen. Es gibt hierfür eine einfache Erklärung: Der Aufrufer legt einen Wert auf dem Stack ab und ruft unsere Funktion über `CALL` auf. `CALL` selbst legt die Rücksprungadresse (RA) auf dem Stack ab und macht einen unbedingten Sprung zur Adresse unserer Funktion. Unsere Funktion hat an jedem Punkt der Ausführung (da sie keine Befehle enthält, die den Stackpointer verändern) das folgende Stacklayout:

- ESP—zeigt auf RA
- ESP+4—zeigt auf die Variable a

Wenn wir andererseits `printf()` aufrufen benötigen wir hier exakt das gleiche Stacklayout mit dem Unterschied des ersten Arguments von `printf()`, welches auf den auszugebenden String zeigt. Unser Code tut hier das Folgende:

Er ersetzt das erste Argument der Funktion mit der Adresse (d.i. dem Pointer) auf den String und springt zu `printf()` als ob wir nicht unsere Funktion `f()`, sondern direkt `printf()` aufrufen würden. Die Funktion `printf()` gibt einen String auf stdout aus und führt dann den RET Befehl aus, der die RA vom Stack holt. Der Control Flow wird nicht an `f()` übergeben, sondern an den Aufrufer von `f()`, womit effektiv die Funktion `f()` umgangen wird.

All dies ist möglich, da `printf()` in allen Fällen ganz am Ende der Funktion `f()` aufgerufen wird. In gewisser Weise besteht Ähnlichkeit zur Funktion `longjmp()`[91]. Natürlich geschieht all dies um die Ausführungsgeschwindigkeit zu erhöhen.

Ein ähnlicher Fall mit dem ARM Compiler wird im Abschnitt „printf() mit mehreren Argumenten“ beschrieben: (**??** on page ??).

[91] wikipedia

OllyDbg

Da dieses Beispiel komliziert ist, untersuchen wir es mit OllyDbg.

OllyDbg kann solche switch() Konstruktionen erkennen und einige nützliche Kommentare hinzufügen. EAX ist zu Beginn auf 2 gesetzt, das entspricht dem Eingabewert der Funktion:

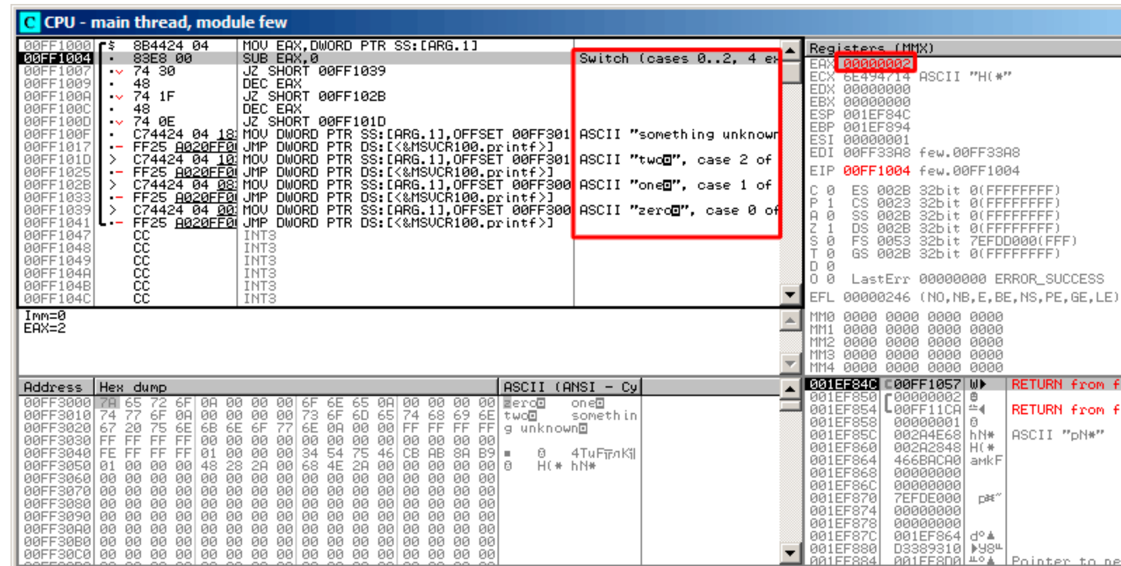


Abbildung 1.42: OllyDbg: EAX enthält jetzt das erste (und einzige) Functionsargument

0 wird in vor der 2 in EAX abgezogen. Natürlich enthält EAX danach immernoch den Wert 2, aber das ZF Flag ist jetzt 0, da das Ergebnis der letzten Berechnung nicht 0 ergeben hat.

Abbildung 1.43: OllyDbg: SUB wurde ausgeführt

DEC wird ausgeführt und EAX enthält nun 1. Da 1 ein von null verschiedener Wert ist, ist das ZF Flag immer noch 0:

Abbildung 1.44: OllyDbg: erstes DEC wurde ausgeführt

Das nächste DEC wird ausgeführt. EAX enthält jetzt 0 und das ZF Flag wird gesetzt, da eine Berechnung 0 ergeben hat.

Abbildung 1.45: OllyDbg: zweites DEC wurde ausgeführt

OllyDbg zeigt an, dass dieser Sprung jetzt getätigt wird.

Ein Pointer auf den String „two“ wird jetzt auf den Stack geschrieben:

Abbildung 1.46: OllyDbg: Pointer auf den String wird an die Stelle des ersten Arguments geschrieben

Man beachte: das aktuelle Argument der Funktion ist 2 und diese 2 befindet sich im Stack nun an der Adresse `0x001EF850`.

MOV schreibt den Pointer auf den String an die Adresse 0x001EF850 (siehe Stackfenster). Dann wird der Sprung ausgeführt. Dies ist der erste Befehl der Funktion printf() in MSVCR100.DLL. (Dieses Beispiel wurde mit der Option /MD kompiliert.)

Abbildung 1.47: OllyDbg: erster Befehl von printf() in MSVCR100.DLL

Die Funktion printf() behandelt den String an der Adresse 0x00FF3010 als (einziges) Argument und gibt ihn aus.

Dies ist der letzte Befehl von `printf()`:

Abbildung 1.48: OllyDbg: letzter Befehl von `printf()` in MSVCR100.DLL

Der String „two“ wurde gerade auf der Konsole ausgegeben.

Wir drücken nun F7 oder F8 (step over) und kehren…nicht zu `f()`, sondern zu `main()` zurück:

Abbildung 1.49: OllyDbg: zurück zu `main()`

Dieser Sprung wird direkt von `printf()` zu `main()` durchgeführt, da RA im Stack nicht auf eine Stelle in `f()`, sondern auf `main()` zeigt. Der Befehl `CALL 0x00FF1000` war der eigentliche Befehl, der `f()` aufgerufen hat.

ARM: Optimierender Keil 6/2013 (ARM Modus)

```
.text:0000014C                  f1:
.text:0000014C 00 00 50 E3        CMP      R0, #0
.text:00000150 13 0E 8F 02        ADREQ    R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A        BEQ      loc_170
.text:00000158 01 00 50 E3        CMP      R0, #1
.text:0000015C 4B 0F 8F 02        ADREQ    R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A        BEQ      loc_170
.text:00000164 02 00 50 E3        CMP      R0, #2
.text:00000168 4A 0F 8F 12        ADRNE    R0, aSomethingUnkno ; "something
    unknown\n"
.text:0000016C 4E 0F 8F 02        ADREQ    R0, aTwo  ; "two\n"
.text:00000170
.text:00000170                  loc_170: ; CODE XREF: f1+8
.text:00000170                           ; f1+14
.text:00000170 78 18 00 EA        B        __2printf
```

Auch hier können wir bei Untersuchung des Code nicht sagen, ob im Quellcode ein switch() oder eine Folge von if()-Ausdrücken vorliegt.

Wir finden hier Befehle mit Prädikaten wieder (wie ADREQ (*Equal*)), welcher nur dann ausgeführt wird, wenn $R0 = 0$ und dann die Adresse des Strings I T«zero\n» nach R0 lädt.

Der folgende BEQ Befehl übergibt den Control Flow an loc_170, falls $R0 = 0$. Ein aufmerksamer Leser könnte sich fragen, ob BEQ korrekt ausgelöst wird, da ADREQ das R0 Register bereits mit einem anderen Wert befüllt hat. Es wird korrekt ausgelöst, da BEQ die Flags, die vom CMP Befehl gesetzt wurden, prüft und ADREQ die Flags nicht verändert.

Die übrigen Befehle kennen wir bereits. Es gibt nur einen Aufruf von printf() am Ende und wir haben diesen Trick bereits hier kennengelernt (**??** on page ??). Am Ende gibt es drei Wege zur Ausführung von printf().

Der letzte Befehl, CMP R0, #2, wird benötigt, um zu prüfen, ob $a = 2$. Wenn dies nicht der Fall ist, lädt ADRNE einen Pointer auf den String *«something unknown \n»* nach R0, da a bereits auf Gleichheit mit 0 oder 1 geprüft wurde und wir können sicher sein, dass die Variable a an dieser Stelle keinen dieser beiden Werte enthält. Falls $R0 = 2$ ist, lädt ADREQ einen Pointer auf den String *«two\n»* nach R0.

ARM: Optimierender Keil 6/2013 (Thumb Modus)

```
.text:000000D4             f1:
.text:000000D4 10 B5       PUSH    {R4,LR}
.text:000000D6 00 28       CMP     R0, #0
.text:000000D8 05 D0       BEQ     zero_case
.text:000000DA 01 28       CMP     R0, #1
.text:000000DC 05 D0       BEQ     one_case
.text:000000DE 02 28       CMP     R0, #2
.text:000000E0 05 D0       BEQ     two_case
.text:000000E2 91 A0       ADR     R0, aSomethingUnkno ; "something
    unknown\n"
.text:000000E4 04 E0       B       default_case

.text:000000E6             zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0       ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0       B       default_case

.text:000000EA             one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0       ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0       B       default_case

.text:000000EE             two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0       ADR     R0, aTwo ; "two\n"
.text:000000F0             default_case ; CODE XREF: f1+10
.text:000000F0                                  ; f1+14
.text:000000F0 06 F0 7E F8 BL      __2printf
.text:000000F4 10 BD       POP     {R4,PC}
```

Wie bereits erwähnt ist es bei den meisten Befehlen im Thumb mode nicht möglich Prädikate für Bedingungen hinzuzufügen, sodass der Thumb-Code hier dem leicht verständlichen x86 CISC-style Code sehr ähnlich ist.

ARM64: Nicht optimierender GCC (Linaro) 4.9

```
.LC12:
        .string "zero"
.LC13:
        .string "one"
.LC14:
        .string "two"
.LC15:
        .string "something unknown"
f12:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     w0, [x29,28]
        ldr     w0, [x29,28]
        cmp     w0, 1
        beq     .L34
        cmp     w0, 2
        beq     .L35
        cmp     w0, wzr
        bne     .L38            ; jump to default label
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        bl      puts
        b       .L32
.L34:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        bl      puts
        b       .L32
.L35:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        bl      puts
        b       .L32
.L38:
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        bl      puts
        nop
.L32:
        ldp     x29, x30, [sp], 32
        ret
```

Der Datentyp des Eingabewertes ist *int*, deshalb wird das Register `W0` anstatt des `X0` Registers verwendet, um ihn aufzunehmen.

Die Pointer auf die Strings werden an `puts()` mit einem ADRP/ADD Befehlspaar übergeben, genauso wie wir es im „Hallo, Welt!“ Beispiel gezeigt haben: 1.5.3 on page 32.

ARM64: Optimierender GCC (Linaro) 4.9

```
f12:
```

```
        cmp     w0, 1
        beq     .L31
        cmp     w0, 2
        beq     .L32
        cbz     w0, .L35
; default case
        adrp    x0, .LC15       ; "something unknown"
        add     x0, x0, :lo12:.LC15
        b       puts
.L35:
        adrp    x0, .LC12       ; "zero"
        add     x0, x0, :lo12:.LC12
        b       puts
.L32:
        adrp    x0, .LC14       ; "two"
        add     x0, x0, :lo12:.LC14
        b       puts
.L31:
        adrp    x0, .LC13       ; "one"
        add     x0, x0, :lo12:.LC13
        b       puts
```

Ein besser optimiertes Stück Code. Der Befehl CBZ (*Compare and Branch on Zero*) springt, falls W0 gleich null ist. Es gibt auch einen direkten Sprung zu puts() anstelle eines Aufrufs, so wie bereits hier erklärt: 1.15.1 on page 178.

MIPS

Listing 1.131: Optimierender GCC 4.4.5 (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; is it 1?
                li      $v0, 1
                beq     $a0, $v0, loc_60
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; is it 2?
                li      $v0, 2
                beq     $a0, $v0, loc_4C
                or      $at, $zero ; branch delay slot, NOP
; jump, if not equal to 0:
                bnez    $a0, loc_38
                or      $at, $zero ; branch delay slot, NOP
; zero case:
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9 ; branch delay slot, NOP
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

loc_38:                                  # CODE XREF: f+1C
                lui     $a0, ($LC3 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
```

```
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "something unknown" ; branch
    delay slot

loc_4C:                                 # CODE XREF: f+14
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

loc_60:                                 # CODE XREF: f+8
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; load delay slot, NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot
```

Die Funktion endet stets mit einem Aufruf von `puts()`, weshalb wir hier einen Sprung zu `puts()` (JR: „Jump Register“) anstelle von „jump and link“ finden. Dieses Feature haben wir bereit in 1.15.1 on page 178 besprochen.

Wir finden auch oft `NOP` Befehle nach `LW` Befehlen. Dies ist „load delay slot“: ein anderer *delay slot* in MIPS. Ein Befehl neben `LW` kann in dem Moment ausgeführt werden, in dem `LW` Werte aus dem Speicher lädt. Der nächste Befehl muss aber nicht das Ergebnis von `LW` verwenden. Moderne MIPS CPUs haben die Eigenschaft abwarten zu können, ob der folgende Befehl das Ergebnis von `LW` verwendet, sodass dieses Vorgehen überholt wirkt, aber GCC fügt für ältere MIPS CPUs immer noch NOPs hinzu. Im Allgemeinen können diese aber ignoriert werden.

Fazit

Eine *switch()* Anweisung mit wenigen Fällen lässt sich nicht von einer *if/else* Konstruktion unterscheiden, zum Beispiel: Listing.1.15.1.

1.15.2 Viele Fälle

Wenn ein `switch()` Ausdruck viele Fälle enthält, ist es für den Compiler nicht günstig sehr großen Code mit vielen JE/JNE Befehlen zu erzeugen.

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
```

```
    default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

Nicht optimierender MSVC

Wir erhalten (MSVC 2010):

Listing 1.132: MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 4
    ja     SHORT $LN1@f
    mov    ecx, DWORD PTR tv64[ebp]
    jmp    DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push   OFFSET $SG739 ; 'zero', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN5@f:
    push   OFFSET $SG741 ; 'one', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN4@f:
    push   OFFSET $SG743 ; 'two', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN3@f:
    push   OFFSET $SG745 ; 'three', 0aH, 00H
    call   _printf
    add    esp, 4
    jmp    SHORT $LN9@f
$LN2@f:
    push   OFFSET $SG747 ; 'four', 0aH, 00H
    call   _printf
```

```
    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad    2 ; Padding für nächstes Label
$LN11@f:
    DD    $LN6@f ; 0
    DD    $LN5@f ; 1
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

Was wir hier sehen ist eine Ansammlung von Aufrufen von `printf()` mit diversen Argumenten. Alle haben nicht Adressen im Speicher des Prozesses, sondern auch interne symbolische Labels, die ihnen vom Compiler zugewiesen werden. Alle diese Labels werden auch in der internen Tabelle `$LN11@f` aufgeführt.

Zu Beginn der Funktion wird der Control Flow an das Label `$LN1@f` abgegeben, wenn a größer ist als 4. An diesem Label wird `printf()` mit dem Argument `'something unknown'` aufgerufen.

Wenn aber der Wert von a kleiner gleich 4 ist, dann wird dieser mit 4 multipliziert und mit der Tabellenadresse `$LN11@f` addiert. Auf diese Weise wird die Adresse innerhalb der Tabelle konstruiert und zeigt genau auf das gewünschte Element. Nehmen wir zum Beispiel an, dass a gleich 2 ist. $2 \cdot 4 = 8$ (alle Tabellenelemente sind Adressen in einem 32-Bit-Prozess und haben daher eine Breite von 4 Bytes). Die Adresse an der Stelle `$LN11@f + 8` ist das Tabellenelement, an dem das Label `$LN4@f` gespeichert ist. JMP holt die Adresse `$LN4@f` aus der Tabelle und springt dorthin.

Diese Tabelle wird manchmal *Jumptable* oder *Verzweigungstabelle* genannt[92].

Dann wird das zugehörige `printf()` mit dem Argument `'two'` aufgerufen.
Der Befehl TTjmp DWORD PTR $LN11@f[ecx*4] bedeutet dabei *springe zum an dieser Stelle gespeicherten DWORD*`$LN11@f + ecx * 4`.

npad (.1.2 on page 683) ist ein Assemblermakro, dass das nächste Label so angeordnet, dass es an einer 4 Byte (oder 16 Bit) Adressgrenze gespeichert wird. Das ist für den Prozessor sehr praktisch, da er die 32-Bit-Werte aus dem Speicher durch den Speicherbus, den Cache, etc. in effektiverer Weise laden kann.

[92]Die ganze Methode wurde in früheren Versionen von Fortran *berechnetes GOTO* genannt: wikipedia. Heutzutage zwar nicht mehr relevant, aber welch ein Ausdruck!

OllyDbg

Untersuchen wir das Beispiel in OllyDbg. Der Eingabewert der Funktion (2) wird nach EAX geladen:

Abbildung 1.50: OllyDbg: das Funktionsargument wird nach EAX geladen

Es wird geprüft, ob der Eingabewert größer als 4 ist. Falls nicht, wird der „default“ Sprung nicht ausgeführt:

Abbildung 1.51: OllyDbg: 2 ist nicht größer als 4: kein Sprung wird ausgeführt

Hier sehen wir eine Jumptable:

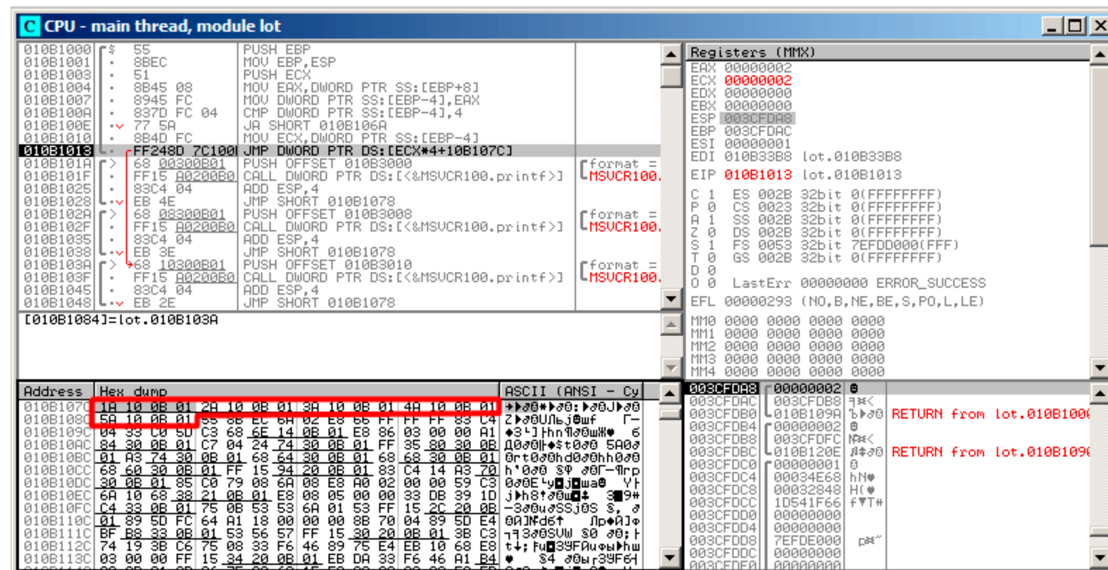


Abbildung 1.52: OllyDbg: Zieladresse mit Jumptable berechnen

Wir haben „Follow in Dump“→ „Address constant“ geklickt, sodass wir jetzt die *Jumptable* im Datenfenster sehen. Hier sind 5 32-Bit-Werte[93]. ECX ist jetzt 2, sodass das zweite Element (beginnend bei null) der Tabelle verwendet wird. Es ist auch möglich durch Klicken auf qFollow in Dump → „Memory address“ in OllyDbg das Element, das durch den JMP Befehl angesteuert wird, anzeigen zu lassen. Dieses Element ist hier `0x010B103A`.

[93]Diese werden von OllyDbg unterstrichen, da es auch FIXUPs sind: 6.5.2 on page 615, wir kommen später darauf zurück

Nach dem Sprung sind wir an der Stelle 0x010B103A: der Code zur Ausgabe von „two“ wird jetzt ausgeführt:

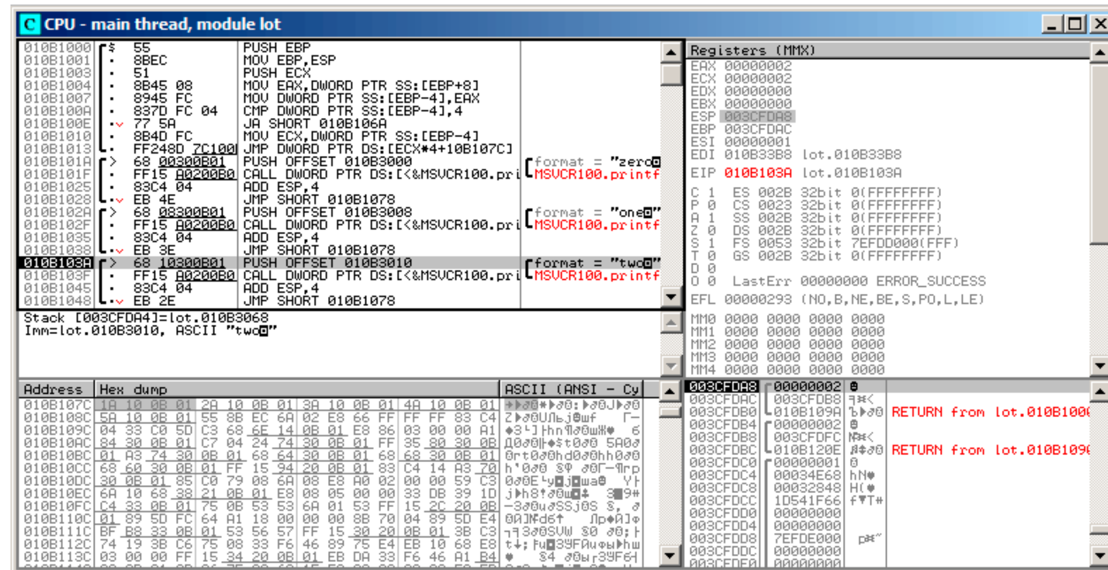


Abbildung 1.53: OllyDbg: jetzt sind wir am *case:* Label

Nicht optimierender GCC

Schauen wir was GCC 4.4.1 erzeugt:

Listing 1.133: GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450
```

```
loc_804840C: ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
        call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

Es ist bis auf eine Nuance das gleiche: das Argument `arg_0` wird mit 4 multipliziert durch eine Verschiebung von 2 Bits nach links (dies entspricht einer Multiplikation mit 4) (1.18.2 on page 253). Dann wird die Adresse des Labels vom `off_804855C` genommen, die in EAX gespeichert wird, und dann wird mit JMP EAX der eigentliche Sprung durchgeführt.

ARM: Optimierender Keil 6/2013 (ARM Modus)

Listing 1.134: Optimierender Keil 6/2013 (ARM Modus)

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable 00000178
    default case
```

```
00000180
00000180             loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA     B       zero_case       ; jumptable 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA     B       one_case        ; jumptable 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA     B       two_case        ; jumptable 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA     B       three_case      ; jumptable 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA     B       four_case       ; jumptable 00000178 case 4

00000194
00000194             zero_case ; CODE XREF: f2+4
00000194                       ; f2:loc_180
00000194 EC 00 8F E2     ADR     R0, aZero       ; jumptable 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8

0000019C
0000019C             one_case ; CODE XREF: f2+4
0000019C                      ; f2:loc_184
0000019C EC 00 8F E2     ADR     R0, aOne        ; jumptable 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF: f2+4
000001A4                      ; f2:loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo        ; jumptable 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF: f2+4
000001AC                        ; f2:loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree      ; jumptable 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8

000001B4
000001B4             four_case ; CODE XREF: f2+4
000001B4                       ; f2:loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF: f2+24
000001B8                       ; f2+2C
000001B8 66 18 00 EA     B       __2printf
```

```
000001BC
000001BC             default_case ; CODE XREF: f2+4
000001BC                          ; f2+8
000001BC D4 00 8F E2     ADR     R0, aSomethingUnkno ; jumptable 00000178
    default case
000001C0 FC FF FF EA     B       loc_1B8
```

Dieser Code verwendet das ARM mode Feature, das alle Befehle eine feste Länge von 4 Byte haben.

Vergessen wir nicht, dass der Maximalwert für a 4 beträgt und jeder größere Wert zur Ausgabe des *«something unknown\n»* Strings führt.

Der erste `CMP R0, #5` Befehl vergleich den Eingabewert a mit 5.

[94] Der nächste `ADDCC PC, PC, R0,LSL#2` Befehl wird nur ausgeführt, falls $R0 < 5$ (*CC=Carry clear / kleiner als*). Wenn `ADDCC` nicht ausgeführt wird (d.h. $R0 \geq 5$), wird ein Sprung zum *default_case* Label ausgeführt.

Aber wenn $R0 < 5$ und `ADDCC` ausgeführt wird, wird das Folgende geschehen:

Der Wert in `R0` wird mit 4 multipliziert. Der Suffix `LSL2` am Befehl steht dabei für „shift left by 2 bits". Aber wie wir später (1.18.2 on page 252) im Abschnitt „Verschiebungen" sehen werden, ist eine Verschiebung um 2 Bits nach links äquivalent zu einer Multiplikation mit 4.

Danach addieren wir $R0 \cdot 4$ zum aktuellen Wert in **PC!** und springen dadurch zu einem der unteren `B` (*Branch*) Befehle.

Im Moment der Ausführung von`ADDCC` ist der Wert von **PC!** (`0x180`) 8 Bytes - oder mit anderen Worten: 2 Befehle - größer als die Adresse, an der sich der `ADDCC` Befehl befindet (`0x178`)

So funktioniert die Pipeline in ARM Prozessoren: wenn `ADDCC` ausgeführt wird, beginnt der Prozessor den Befehl nach dem nächsten abzuarbeiten und deshalb zeigt **PC!** hierher. Das müssen wir im Kopf behalten.

Wenn $a = 0$, dann wird dies zum Wert in **PC!** addiert und der aktuelle Wert des **PC!** wird nach **PC!** geschrieben (welcher 8 Byte größer ist) und es wird zum Label *loc_180* gesprungen, welches 8 Byte größer ist als die Adresse des `ADDCC` Befehls.

Wenn $a = 1$, dann wird $PC + 8 + a \cdot 4 = PC + 8 + 1 \cdot 4 = PC + 12 = 0x184$ nach **PC!** geschrieben,was der Adresse des *loc_184* Labels entspricht.

Jedes Mal wenn a um 1 erhöht wird, erhöht sich der **PC!** um 4.

Dabei ist 4 die Länge eines Befehls im ARM mode und auch die Länge jedes `B` Befehls, von denen sich hier 5 befinden.

Jeder dieser fünf `B` Befehle gibt den Control Flow weiter so wie es im *switch()* Ausdruck programmiert wurde.

Hier werden jeweils die Pointer auf die zugehörigen Strings geladen, etc.

[94]ADD—Addition

ARM: Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.135: Optimierender Keil 6/2013 (Thumb Modus)

```
000000F6                         EXPORT f2
000000F6                     f2
000000F6 10 B5                   PUSH    {R4,LR}
000000F8 03 00                   MOVS    R3, R0
000000FA 06 F0 69 F8             BL      __ARM_common_switch8_thumb ; switch 6
    cases

000000FE 05                      DCB 5
000000FF 04 06 08 0A 0C 10       DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for
    switch statement
00000105 00                      ALIGN 2
00000106
00000106                     zero_case ; CODE XREF: f2+4
00000106 8D A0                   ADR     R0, aZero ; jumptable 000000FA case 0
00000108 06 E0                   B       loc_118

0000010A
0000010A                     one_case ; CODE XREF: f2+4
0000010A 8E A0                   ADR     R0, aOne ; jumptable 000000FA case 1
0000010C 04 E0                   B       loc_118

0000010E
0000010E                     two_case ; CODE XREF: f2+4
0000010E 8F A0                   ADR     R0, aTwo ; jumptable 000000FA case 2
00000110 02 E0                   B       loc_118

00000112
00000112                     three_case ; CODE XREF: f2+4
00000112 90 A0                   ADR     R0, aThree ; jumptable 000000FA case 3
00000114 00 E0                   B       loc_118

00000116
00000116                     four_case ; CODE XREF: f2+4
00000116 91 A0                   ADR     R0, aFour ; jumptable 000000FA case 4
00000118
00000118                     loc_118 ; CODE XREF: f2+12
00000118                             ; f2+16
00000118 06 F0 6A F8             BL      __2printf
0000011C 10 BD                   POP     {R4,PC}

0000011E
0000011E                     default_case ; CODE XREF: f2+4
0000011E 82 A0                   ADR     R0, aSomethingUnkno ; jumptable
    000000FA default case
00000120 FA E7                   B       loc_118


000061D0                         EXPORT __ARM_common_switch8_thumb
000061D0                     __ARM_common_switch8_thumb ; CODE XREF:
    example6_f2+4
000061D0 78 47                   BX      PC
```

```
000061D2 00 00                ALIGN 4
000061D2                    ; End of function __ARM_common_switch8_thumb
000061D2
000061D4                    __32__ARM_common_switch8_thumb ; CODE XREF:
    __ARM_common_switch8_thumb
000061D4 01 C0 5E E5          LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1          CMP     R3, R12
000061DC 0C 30 DE 27          LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37          LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0          ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1          BX      R12
000061E8                    ; End of function __32__ARM_common_switch8_thumb
```

Man kann sich nicht sicher sein, dass alle Befehle im Thumb und Thumb-2 mode dieselbe Größe haben. Man kann sogar sagen, dass die Befehle hier genau wie in x86 variable Längen haben.

Deshalb wird hier eine spezielle Tabelle verwendet, die Informationen darüber enthält wie viele Fälle vorliegen (ohne den Default-Case) und es wird für jeden Fall ein Label mit einem Offset für den Control Flow im zugehörigen Fall angegeben.

Hier taucht eine spezielle Funktion namens *__ARM_common_switch8_thumb* auf, die mit der Tabelle und der Übergabe des Control Flows umgeht. Sie beginnt mit `BX PC`, dessen Aufgabe es ist, den Prozessor in den ARM mode zu versetzen. Danach finden wir die Funktion für den Umgang mit der Tabelle.

Es ist hier zu fortgeschritten um weiter ins Details zu gehen, daher lassen wir es für den Moment hierbei bewenden.

Ist ist interessant festzustellen, dass die Funktion das LR Register als Pointer auf die Tabelle verwendet.

Tatsächlich enthält LR nach dem Aufruf der Funktion die Adresse nach dem Befehl `BL __ARM_common_switch8_thumb`, an dem die Tabelle beginnt.

Es ist auch bemerkenswert, dass der Code als eine separate Funktion erzeugt wird, um wiederverwendet werden zu können, sodass der Compiler nicht für jeden switch() Ausdruck den gleichen Code erzeugen muss.

IDA hat erfolgreich ermittelt, dass es sich um eine Servicefunktion und eine Tabelle handelt und hat Kommentare wie etwa `jumptable 000000FA case 0` zu den Labels hinzugefügt.

MIPS

Listing 1.136: Optimierender GCC 4.4.5 (IDA)

```
f:
                lui     $gp, (__gnu_local_gp >> 16)
; springe zu loc_24 , falls der Eingabewert kleiner als 5 ist:
                sltiu   $v0, $a0, 5
                bnez    $v0, loc_24
                la      $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
; Eingabewert ist größer gleich 5.
```

```
; "something unknown" ausgeben und beenden:
                lui     $a0, ($LC5 >> 16)  # "something unknown"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC5 & 0xFFFF)  # "something unknown" ; branch
    delay slot

loc_24:                                   # CODE XREF: f+8
; lade Adresse der Jumptable
; LA ist ein Pseudo-Befehl, der für ein LUI und ADDIU Paar steht:
                la      $v0, off_120
; multipliere Eingabewert mit 4:
                sll     $a0, 2
; multiplizierten Wert und Adresse der Jumptable addieren:
                addu    $a0, $v0, $a0
; lade Element aus Jumptable:
                lw      $v0, 0($a0)
                or      $at, $zero ; NOP
; Sprung zur Adresse in der Jumptable:
                jr      $v0
                or      $at, $zero ; branch delay slot, NOP

sub_44:                                   # DATA XREF: .rodata:0000012C
; "three" ausgeben und beenden
                lui     $a0, ($LC3 >> 16)  # "three"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC3 & 0xFFFF)  # "three" ; branch delay slot

sub_58:                                   # DATA XREF: .rodata:00000130
; "four" ausgeben und beenden
                lui     $a0, ($LC4 >> 16)  # "four"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC4 & 0xFFFF)  # "four" ; branch delay slot

sub_6C:                                   # DATA XREF: .rodata:off_120
; "zero" ausgeben und beenden
                lui     $a0, ($LC0 >> 16)  # "zero"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC0 & 0xFFFF)  # "zero" ; branch delay slot

sub_80:                                   # DATA XREF: .rodata:00000124
; "one" ausgeben und beenden
                lui     $a0, ($LC1 >> 16)  # "one"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC1 & 0xFFFF)  # "one" ; branch delay slot
```

```
sub_94:                                 # DATA XREF: .rodata:00000128
; "two" ausgeben und beenden
                lui     $a0, ($LC2 >> 16)  # "two"
                lw      $t9, (puts & 0xFFFF)($gp)
                or      $at, $zero ; NOP
                jr      $t9
                la      $a0, ($LC2 & 0xFFFF)  # "two" ; branch delay slot

; kann im .rodata Segment abgelegt werden:
off_120:        .word sub_6C
                .word sub_80
                .word sub_94
                .word sub_44
                .word sub_58
```

Der für uns neue Befehl ist SLTIU („Set on Less Than Immediate Unsigned“).

Dies ist das gleiche wie SLTU („Set on Less Than Unsigned“); das „I“ steht dabei für „immediate“, d.h. für den Befehl muss eine Zahl angegeben werden.

BNEZ ist „Branch if Not Equal to Zero“.

Der Code ist den anderen ISAs sehr ähnlich. SLL („Shift Word Left Logical“) führt eine Multiplikation mit 4 durch. Da MIPS eine 32-Bit CPU ist, sind auch die Adressen in der *Jumptable* 32 Bit groß.

Fazit

Das grobe Gerüst eines *switch()*:

Listing 1.137: x86

```
MOV REG, input
CMP REG, 4 ; maximale Anzahl von Fällen
JA default
SHL REG, 2 ; finde Element in der Tabelle. 3 Bits schieben in x64.
MOV REG, jump_table[REG]
JMP REG

case1:
    ; beliebiger Code
    JMP exit
case2:
    ; beliebiger Code
    JMP exit
case3:
    ; beliebiger Code
    JMP exit
case4:
    ; beliebiger Code
    JMP exit
case5:
    ; beliebiger Code
```

```
    JMP exit

default:

    ...

exit:

    ....

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5
```

Der Sprung zur Adresse in der Jumptable kann auch durch den folgenden Befehl realisiert werden:
`JMP jump_table[REG*4]` oder `JMP jump_table[REG*8]` in x64.

Eine *Jumptable* ist nur ein Array von Pointern, genau wie das hier beschriebene: 1.20.5 on page 332.

1.15.3 Wenn es mehrere *case* Ausdrücke in einem Block gibt

Hier ist eine weit verbreitete Konstruktion: mehrere *case* Ausdrücke für einen einzigen Block:

```
#include <stdio.h>

void f(int a)
{
        switch (a)
        {
        case 1:
        case 2:
        case 7:
        case 10:
                printf ("1, 2, 7, 10\n");
                break;
        case 3:
        case 4:
        case 5:
        case 6:
                printf ("3, 4, 5\n");
                break;
        case 8:
        case 9:
        case 20:
        case 21:
                printf ("8, 9, 21\n");
                break;
        case 22:
```

```
                printf ("22\n");
                break;
        default:
                printf ("default\n");
                break;
        };
};

int main()
{
        f(4);
};
```

Es ist zu verschwenderisch einen Block für jeden möglichen Fall zu erzeugen, sodass normalerweise ein Block und eine Art Dispatcher erzeugt werden.

MSVC

Listing 1.138: Optimierender MSVC 2010

```
$SG2798 DB      '1, 2, 7, 10', 0aH, 00H
$SG2800 DB      '3, 4, 5', 0aH, 00H
$SG2802 DB      '8, 9, 21', 0aH, 00H
$SG2804 DB      '22', 0aH, 00H
$SG2806 DB      'default', 0aH, 00H

_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        dec     eax
        cmp     eax, 21
        ja      SHORT $LN1@f
        movzx   eax, BYTE PTR $LN10@f[eax]
        jmp     DWORD PTR $LN11@f[eax*4]
$LN5@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
        jmp     DWORD PTR __imp__printf
$LN4@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
        jmp     DWORD PTR __imp__printf
$LN3@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
        jmp     DWORD PTR __imp__printf
$LN2@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
        jmp     DWORD PTR __imp__printf
$LN1@f:
        mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
        jmp     DWORD PTR __imp__printf
        npad    2 ; $LN11@f Tabelle auf 16-Byte-Grenze bringen
$LN11@f:
        DD      $LN5@f ; '1, 2, 7, 10' ausgeben
        DD      $LN4@f ; '3, 4, 5' ausgeben
```

```
34          DD      $LN3@f ; '8, 9, 21' ausgeben
35          DD      $LN2@f ; '22' ausgeben
36          DD      $LN1@f ; 'default' ausgeben
37  $LN10@f:
38          DB      0 ; a=1
39          DB      0 ; a=2
40          DB      1 ; a=3
41          DB      1 ; a=4
42          DB      1 ; a=5
43          DB      1 ; a=6
44          DB      0 ; a=7
45          DB      2 ; a=8
46          DB      2 ; a=9
47          DB      0 ; a=10
48          DB      4 ; a=11
49          DB      4 ; a=12
50          DB      4 ; a=13
51          DB      4 ; a=14
52          DB      4 ; a=15
53          DB      4 ; a=16
54          DB      4 ; a=17
55          DB      4 ; a=18
56          DB      4 ; a=19
57          DB      2 ; a=20
58          DB      2 ; a=21
59          DB      3 ; a=22
60  _f      ENDP
```

Wir sehen hier zwei Tabellen: die erste Tabelle (`$LN10@f`) ist eine Indextabelle und die zweite (`$LN11@f`) ist ein Array von Pointern auf Blöcke.

Zuerst wird der Eingabewert als Index in der Indextabelle verwendet (Zeile 13).

Hier ist eine kurze Legende für die Werte in der Tabelle: 0 ist der erste *case* Block (für die Werte 1, 2, 7, 10), 1 ist der zweite (für die Werte 3, 4, 5), 2 ist der dritte (für die Werte 8, 9, 21), 3 ist der vierte (für die Werte 22), 4 ist der Defaultblock. Hier erhalten wir einen Index für die zweite Tabelle aus Pointern und springen zu einem solchen (Zeile 14).

Bemerkenswert ist auch, dass es keinen Case für den Eingabewert 0 gibt.

Aus diesem Grund haben wir den `DEC` Befehl in Zeile 10 und die Tabelle beginnt bei $a = 1$, da kein Tabellenelement für $a = 0$ angelegt werden muss.

Dies ist ein weitverbreitetes Muster.

Warum ist dieses Vorgehen so ökonomisch? Warum ist es nicht möglich wie vorher in (1.15.2 on page 197) vorzugehen mit nur einer Tabelle aus Blockpointern? Der Grund hierfür ist, dass die Elemente in der Indextabelle nur 8 Bit groß sind und alles deshalb deutlich kompakter ist.

GCC

GCC erledigt seinen Job unter Verwendung von nur einer Pointertabelle wie bereits hier besprochen (1.15.2 on page 197).

ARM64: Optimierender GCC 4.9.1

Es wird kein Code ausgeführt, wenn der Eingabewert 0 ist, weshalb GCC versucht, die Jumptable kleiner zu machen und erst beim Eingabewert 1 zu beginnen.

GCC 4.9.1 für ARM64 verwendeten einen noch ausgefeilteren Trick. Es ist möglich alle Offsets als 8 Bit Werte zu kodieren.

Erinnern wir uns, dass alle ARM64 Befehle eine Größe von 4 Bytes haben.

GCC verwendet den Umstand, dass alle Offsets in unserem Minimalbeispiel in der Nähe voneinander liegen. Daher kann die Jumptable aus einzelnen Bytes bestehen.

Listing 1.139: Optimierender GCC 4.9.1 ARM64

```
f14:
; Eingabewert in W0
        sub     w0, w0, #1
        cmp     w0, 21
; verzweige, falls kleiner gleich (vorzeichenlos):
        bls     .L9
.L2:
; "default" ausgeben:
        adrp    x0, .LC4
        add     x0, x0, :lo12:.LC4
        b       puts
.L9:
; lade Jumptableadresse von X1:
        adrp    x1, .L4
        add     x1, x1, :lo12:.L4
; W0=Eingabewert-1
; lade Byte aus der Tabelle:
        ldrb    w0, [x1,w0,uxtw]
; lade Adresse des Lrtx Labels:
        adr     x1, .Lrtx4
; multipliziere Tabellenelement mit 4 (durch Schieben von 2 Bits nach links)
    und addiere (oder subtrahiere) es zur
Adresse von Lrtx:
        add     x0, x1, w0, sxtb #2
; springe zur berechneten Adresse:
        br      x0
; dieses Label zeigt auf das Code (Text) Segment:
.Lrtx4:
        .section        .rodata
; alles nach dem ".section" Ausdruck wird im Read-only (rodata) Segment
    angelegt:
.L4:
        .byte   (.L3 - .Lrtx4) / 4      ; case 1
        .byte   (.L3 - .Lrtx4) / 4      ; case 2
        .byte   (.L5 - .Lrtx4) / 4      ; case 3
```

```
        .byte   (.L5 - .Lrtx4) / 4      ; case 4
        .byte   (.L5 - .Lrtx4) / 4      ; case 5
        .byte   (.L5 - .Lrtx4) / 4      ; case 6
        .byte   (.L3 - .Lrtx4) / 4      ; case 7
        .byte   (.L6 - .Lrtx4) / 4      ; case 8
        .byte   (.L6 - .Lrtx4) / 4      ; case 9
        .byte   (.L3 - .Lrtx4) / 4      ; case 10
        .byte   (.L2 - .Lrtx4) / 4      ; case 11
        .byte   (.L2 - .Lrtx4) / 4      ; case 12
        .byte   (.L2 - .Lrtx4) / 4      ; case 13
        .byte   (.L2 - .Lrtx4) / 4      ; case 14
        .byte   (.L2 - .Lrtx4) / 4      ; case 15
        .byte   (.L2 - .Lrtx4) / 4      ; case 16
        .byte   (.L2 - .Lrtx4) / 4      ; case 17
        .byte   (.L2 - .Lrtx4) / 4      ; case 18
        .byte   (.L2 - .Lrtx4) / 4      ; case 19
        .byte   (.L6 - .Lrtx4) / 4      ; case 20
        .byte   (.L6 - .Lrtx4) / 4      ; case 21
        .byte   (.L7 - .Lrtx4) / 4      ; case 22
        .text
; alles nach dem ".text" Ausdruck wird im Code (Text) Segment angelegt:
.L7:
; "22" ausgeben
        adrp    x0, .LC3
        add     x0, x0, :lo12:.LC3
        b       puts
.L6:
; "8, 9, 21" ausgeben
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        b       puts
.L5:
; "3, 4, 5" ausgeben
        adrp    x0, .LC1
        add     x0, x0, :lo12:.LC1
        b       puts
.L3:
; "1, 2, 7, 10" ausgeben
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        b       puts
.LC0:
        .string "1, 2, 7, 10"
.LC1:
        .string "3, 4, 5"
.LC2:
        .string "8, 9, 21"
.LC3:
        .string "22"
.LC4:
        .string "default"
```

Kompilieren wir dieses Beispiel in eine Object-Datei und öffnen es ist IDA. Hier ist die

Jumptable:

Listing 1.140: jumptable in IDA

```
.rodata:0000000000000064           AREA .rodata, DATA, READONLY
.rodata:0000000000000064           ; ORG 0x64
.rodata:0000000000000064 $d        DCB    9    ; case 1
.rodata:0000000000000065           DCB    9    ; case 2
.rodata:0000000000000066           DCB    6    ; case 3
.rodata:0000000000000067           DCB    6    ; case 4
.rodata:0000000000000068           DCB    6    ; case 5
.rodata:0000000000000069           DCB    6    ; case 6
.rodata:000000000000006A           DCB    9    ; case 7
.rodata:000000000000006B           DCB    3    ; case 8
.rodata:000000000000006C           DCB    3    ; case 9
.rodata:000000000000006D           DCB    9    ; case 10
.rodata:000000000000006E           DCB 0xF7    ; case 11
.rodata:000000000000006F           DCB 0xF7    ; case 12
.rodata:0000000000000070           DCB 0xF7    ; case 13
.rodata:0000000000000071           DCB 0xF7    ; case 14
.rodata:0000000000000072           DCB 0xF7    ; case 15
.rodata:0000000000000073           DCB 0xF7    ; case 16
.rodata:0000000000000074           DCB 0xF7    ; case 17
.rodata:0000000000000075           DCB 0xF7    ; case 18
.rodata:0000000000000076           DCB 0xF7    ; case 19
.rodata:0000000000000077           DCB    3    ; case 20
.rodata:0000000000000078           DCB    3    ; case 21
.rodata:0000000000000079           DCB    0    ; case 22
.rodata:000000000000007B ; .rodata ends
```

Im Fall von 1, 9 wird also mit 4 multipliziert und zur Adresse des Labels `Lrtx4` addiert.

Im Fall von 22, wird 0 mit 4 multipliziert; mit dem Ergebnis 0.

Direkt hinter dem `Lrtx4` Label befindet sich das L7 Label, an dem sich der Code befindet, der „22“ ausgibt.

Es gibt keine Jumptable im Codesegment; sie wird in einem getrennten .rodata Segment angelegt (es besteht keine Notwendigkeit, die Tabelle im Codesegment anzulegen).

Hier befinden sich auch negative Bytes (0xF7), die für das Zurückspringen im Code verwendet werden, um den „Default“ String am Label `.L2` auszugeben.

1.15.4 Fallthrough

Eine andere übliche Verwendung des `switch()` Operators ist der sogenannte „Fallthrough“. Hier ist ein einfaches Beispiel[95]:

```
bool is_whitespace(char c) {
        switch (c) {
                case ' ': // fallthrough
```

[95] Kopiert von https://github.com/azonalon/prgraas/blob/master/prog1lib/lecture_examples/is_whitespace.c

```
                case '\t': // fallthrough
                case '\r': // fallthrough
                case '\n':
                        return true;
                default: // not whitespace
                        return false;
        }
}
```

Ein etwas komplizierteres Beispiel aus dem Linux Kernel[96]:

```
char nco1, nco2;

void f(int if_freq_khz)
{

        switch (if_freq_khz) {
                default:
                        printf("IF=%d KHz is not supportted, 3250 assumed\n↙
    ↳ ", if_freq_khz);
                        /* fallthrough */
                case 3250: /* 3.25Mhz */
                        nco1 = 0x34;
                        nco2 = 0x00;
                        break;
                case 3500: /* 3.50Mhz */
                        nco1 = 0x38;
                        nco2 = 0x00;
                        break;
                case 4000: /* 4.00Mhz */
                        nco1 = 0x40;
                        nco2 = 0x00;
                        break;
                case 5000: /* 5.00Mhz */
                        nco1 = 0x50;
                        nco2 = 0x00;
                        break;
                case 5380: /* 5.38Mhz */
                        nco1 = 0x56;
                        nco2 = 0x14;
                        break;
        }
};
```

Listing 1.141: Optimizing GCC 5.4.0 x86

```
.LC0:
        .string "IF=%d KHz is not supportted, 3250 assumed\n"
f:
        sub     esp, 12
        mov     eax, DWORD PTR [esp+16]
```

[96]Kopiert von https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c

```
        cmp     eax, 4000
        je      .L3
        jg      .L4
        cmp     eax, 3250
        je      .L5
        cmp     eax, 3500
        jne     .L2
        mov     BYTE PTR nco1, 56
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L4:
        cmp     eax, 5000
        je      .L7
        cmp     eax, 5380
        jne     .L2
        mov     BYTE PTR nco1, 86
        mov     BYTE PTR nco2, 20
        add     esp, 12
        ret
.L2:
        sub     esp, 8
        push    eax
        push    OFFSET FLAT:.LC0
        call    printf
        add     esp, 16
.L5:
        mov     BYTE PTR nco1, 52
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L3:
        mov     BYTE PTR nco1, 64
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
.L7:
        mov     BYTE PTR nco1, 80
        mov     BYTE PTR nco2, 0
        add     esp, 12
        ret
```

Wir gelangen zum `.L5` Label, wenn die Eingabe der Funktion die Zahl 3250 ist. Aber wir können dieses Label von der anderen Seite erreichen: wir sehen, dass es keine Sprünge zwischen dem Aufruf von `printf()` und dem `.L5` Label gibt.

Jetzt verstehen wir auch, warum der *switch()* Ausdruck manchmal eine Quelle von Bugs ist: ein einziges vergessenes *break* verändert einen *switch()* Ausdruck in einen *Fallthrough* und mehrere Blöcke anstelle eine einzigen werden ausgeführt.

1.15.5 Übungen

Übung #1

Der C-Code des Beispiels in 1.15.2 on page 191 soll so neu geschrieben werden, dass der Compiler die gleiche Funktionalität in noch kürzerem Code erreichen kann.

1.16 Schleifen

1.16.1 Einfaches Beispiel

x86

Es gibt einen speziellen LOOP Befehl im x86 Befehlssatz, der den Wert des Registers ECX prüft und falls dieser ungleich 0 ist, dekrementiert und danach die den control flow wieder an das Label des LOOP Operanden übergibt. Vermutlich ist dieser Befehl nicht allzu geläufig und es gibt keine modernen Compiler, welche ihn automatisch erzeugen. Wenn man also diesen Befehl irgendwo im Code entdeckt, dann ist es äußerst wahrscheinlich, dass es sich um ein handgeschriebenes Stück Assemblercode handelt.

In C/C++ werden Schleifen normalerweise mittels `for()`-, `while()`- oder `do/while()`-Ausdrücken erzeugt.

Starten wir mit `for()`.

Dieser Ausdruck definiert eine Schleifeninitialisierung (setzt den Zähler auf einen Startwert), definiert eine Schleifenbedingung (ist der Zähler größer als ein Grenzwert?), legt fest, was in jedem Durchlauf (Inkrement/Dekrement) geschieht und umschließt einen Schleifenkörper.

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

Der erzeugte Code besteht ebenfalls aus vier Teilen.

Beginnen wir mit einem einfachen Beispiel:

```
#include <stdio.h>

void printing_function(int i)
{
        printf ("f(%d)\n", i);
};

int main()
{
        int i;

        for (i=2; i<10; i++)
                printing_function(i);
```

```
        return 0;
};
```

Ergebnis (MSVC 2010):

Listing 1.142: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; Schleife wird initialisiert
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; hier steht, was nach jedem Durchlauf
    getan
    wird:
    add    eax, 1                  ; addiere 1 zum (i) Wert
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; diese Bedingung wird vor jedem Durchlauf
    geprüft.
    jge    SHORT $LN1@main         ; wenn (i) größer oder gleich 10 ist,
    beende Schleife loop
    mov    ecx, DWORD PTR _i$[ebp] ; Schleifenkörper: call
    printing_function(i)
    push   ecx
    call   _printing_function
    add    esp, 4
    jmp    SHORT $LN2@main         ; Sprung zum Anfang der Schleife
$LN1@main:                         ; Ende der Schleife
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Hier gibt es nichts Besonderes zu sehen.

GCC 4.4.1 erzeugt einen fast identischen Code mit nur einen kleinen Unterschied:

Listing 1.143: GCC 4.4.1

```
main            proc near

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; (i) initialisieren
```

```
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    printing_function
                add     [esp+20h+var_4], 1  ; (i) erhöhen

loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; falls i<=9, Schleife fortsetzen
                mov     eax, 0
                leave
                retn
main            endp
```

Schauen wir uns nun an, was wir erhalten, wenn wir die Optimierung aktivieren (/Ox):

Listing 1.144: Optimierender MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _printing_function
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP
```

Was hier passiert ist, dass der Speicherplatz für die i Variable nicht mehr auf dem lokalen Stack bereitgestellt wird, sondern das extra ein Register, ESI, hierfür verwendet wird. Dies ist bei derartig kleinen Funktionen möglich, wenn nicht zu viele lokalen Variablen existieren.

Wichtig ist, dass die f() Funktion den Wert im Register ESI nicht verändern darf. Unser Compiler ist sich dieser Sache hier sicher. Und falls der Compiler entscheidet, das ESI auch innerhalb der Funktion f() zu verwenden, würde der Wert des Registers im Funktionsprolog gesichert und im Funktionsepilog wiederhergestellt werden; fast genauso wie im folgenden Listing. Man beachte das PUSH ESI/POP ESI bei Funktionsbeginn und -ende.

Probieren wir aus, was GCC 4.4.1 mit maximaler Optimierung (-O3 option) liefert:

Listing 1.145: Optimierender GCC 4.4.1

```
main            proc near

var_10          = dword ptr -10h
```

```
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    printing_function
                mov     [esp+10h+var_10], 3
                call    printing_function
                mov     [esp+10h+var_10], 4
                call    printing_function
                mov     [esp+10h+var_10], 5
                call    printing_function
                mov     [esp+10h+var_10], 6
                call    printing_function
                mov     [esp+10h+var_10], 7
                call    printing_function
                mov     [esp+10h+var_10], 8
                call    printing_function
                mov     [esp+10h+var_10], 9
                call    printing_function
                xor     eax, eax
                leave
                retn
main            endp
```

Aha, GCC hat unsere Schleife unrolled (d.h. ausgerollt).

Loop unwinding hat Vorteile in Fällen, in denen es nicht viele Schleifendurchläufe gibt und Ausführungszeit durch das Weglassen der Befehle für die Kontrollstrukturen der Schleife gewonnen werden kann. Andererseits ist der erzeugte Code natürlich deutlich länger.

Große Schleifen zu *unrollen* ist heutzutage nicht empfehlenswert, denn größere Funktionen erfordern einen größeren Cache-Fußabdruck.[97].

Gut, nun wollen wir den Höchstwert der Variable i auf 100 setzen und kompilieren erneut. GCC liefert:

Listing 1.146: GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2    ; i=2
```

[97]Ein hervorragender Artikel zum Thema: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)][98]. Für weitere Empfehlungen von Intel zum Unrolling siehe hier: [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)3.4.1.7].

```
                sub     esp, 1Ch

; Label loc_80484D0 (Schleifenkörperbeginn) auf 16-Byte-Grenze setzen:
                nop

loc_80484D0:
; übergebe (i) als erstes Argument von printing_function():
                mov     [esp+20h+var_20], ebx
                add     ebx, 1    ; i++
                call    printing_function
                cmp     ebx, 64h  ; i==100?
                jnz     short loc_80484D0 ; wenn nicht, fortsetzen
                add     esp, 1Ch
                xor     eax, eax  ; return 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

Das Ergebnis ist sehr ähnlich dem, das MSVC 2010 mit Optimierung (/Ox) erzeugt, mit der Ausnahme, dass das EBX Register für die Variable i verwendet wird.

GCC ist sicher, dass das Register innerhalb der `f()` Funktion nicht verändert wird und sollte dies doch der Fall sein, dass es im Funktionsprolog gesichert und im Funktionsepilog wiederhergestellt werden wird, genau wie hier in der `main()` Funktion.

x86: OllyDbg

Wir kompilieren unser Beispiel in MSVC 2010 mit den Optionen `/Ox` und `/Ob0` und laden es in OllyDbg.

Es scheint, dass OllyDbg in der Lage ist, einfache Schleifen zu erkennen und in eckigen Klammern darzustellen, um die Übersichtlichkeit zu erhöhen:

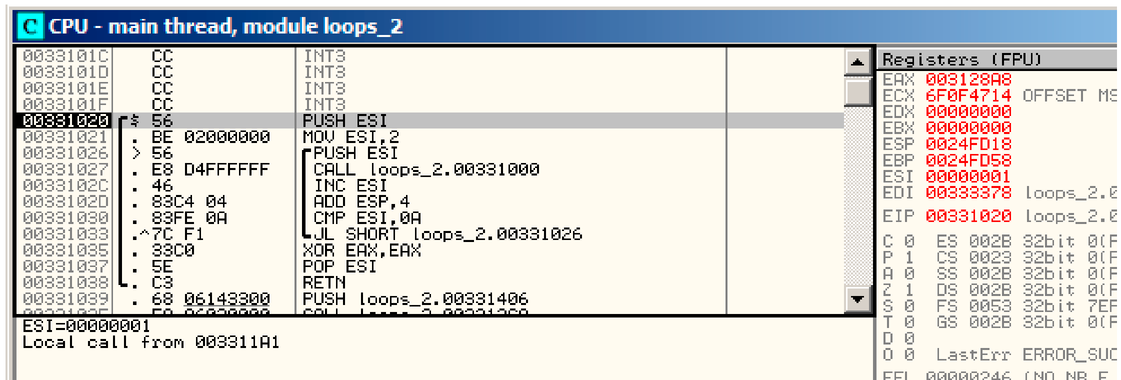


Abbildung 1.54: OllyDbg: `main()` Einstieg

Verfolgen mit (F8 — step over) zeigt ESI incrementing. Hier ist zum Beispiel, $ESI = i = 6$:

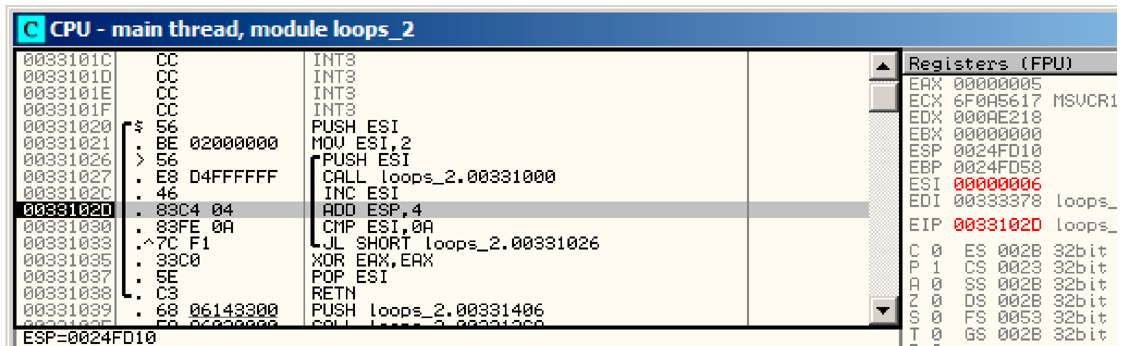


Abbildung 1.55: OllyDbg: Schleifenkörper wird gerade ausgeführt für $i = 6$

9 ist der letzte Wert in der Schleife Deshalb triggert JL nach der inkrement-Anweisung nicht und die Funktion wird beendet:

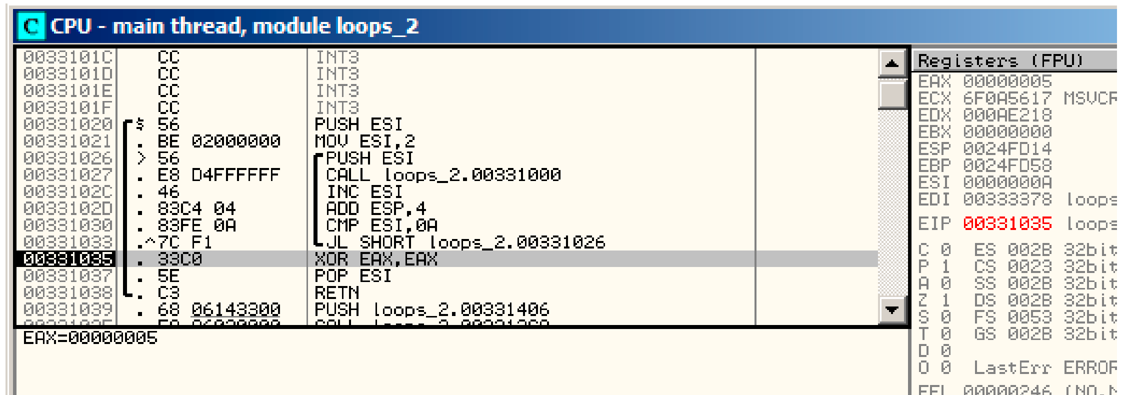


Abbildung 1.56: OllyDbg: $ESI = 10$, Ende der Schleife

x86: tracer

Wie wir bemerken ist es nicht sonderlich komfortabel, Werte im Debugger manuell nachzuverfolgen. Aus diesem Grund probieren wir tracer aus.

Wir öffnen das kompilierte Beispiel in IDA, finden die Adresse mit dem Befehl PUSH ESI (das einzige Argument an f() übergebend), welche hier 0x401026 ist und aktivieren den tracer:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX setzt einen Breakpoint an der Adresse und der Tracer zeigt uns den momentanen Status der Register an. In tracer.log sehen wir das Folgende:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
```

```
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

Wir sehen wie der Wert des ESI Registers sich schrittweise von 2 zu 9 verändert.

Mehr noch, der tracer kann alle Registerwerte für alle Adressen innerhalb der Funktion zusammensammeln. Dies wird hier mit *trace* (dt. Nachverfolgung) bezeichnet. Jeder Befehl wird verfolgt, alle interessanten Registerwerte werden aufgezeichnet.

Danach die ein IDA.idc-script erzeugt, das Kommentare hinzufügt. Wir haben also herausgefunden, dass die Adresse der `main()` Funktion `0x00401020` ist und wir führen nun das Folgende aus:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF setzt einen Breakpoint auf eine Funktion.

Als Ergebnis erahlten wir die Skripte `loops_2.exe.idc` und `loops_2.exe_clear.idc`.

Wir laden `loops_2.exe.idc` in IDA und erhalten:

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Abbildung 1.57: IDA mit geladenem .idc-script

Wir sehen, dass der Wert von ESI zu Beginn der Schleife zwischen 2 und 9 und nach dem Inkrement zwischen 3 und 0xA (10) liegt. Wir sehen auch, dass die Funktion `main()` mit dem Rückgabewert 0 in EAX terminiert.

tracer erzeugt ebenfalls die Datei `loops_2.exe.txt`, welche Informationen darüber enthält, welcher Befehl wie oft ausgeführt wurde, sowie zugehörige Registerwerte:

Listing 1.147: loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested maximum ↙
    ↳ level (1) reached, skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

An dieser Stelle können wir grep verwenden.

ARM

Nicht optimierender Keil 6/2013 (ARM Modus)

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF: main+1C
        MOV     R0, R4
        BL      printing_function
        ADD     R4, R4, #1

loc_368  ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Der Zähler i wird im Register R4 gespeichert. Der Befehl `MOV R4, #2` initialisiert i. Die Befehle `MOV R0, R4` und `BL printing_function` bilden den Körper der Schleife; der erste Befehl bereitet das Argument für die `f()` Funktion vor und der zweite ruft die Funktion auf.

Der Befehl `ADD R4, R4, #1` erhöht die i Variable in jedem Durchlauf um 1. `CMP R4, #0xA` vergleicht i mit `0xA` (10). Der nächste Befehl BLT (*Branch Less Than*) springt, falls i kleiner als 10 ist. Sonst wird 0 in das Register R0 geschrieben (unsere Funktion liefert den Wert 0 zurück) und die Funktionsausführung wird beendet.

Optimierender Keil 6/2013 (Thumb Modus)

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      printing_function
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Praktisch das gleiche.

Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
_main
```

```
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

In meiner `f()` Funktion befand sich tatsächlich Folgendes:

```
void printing_function(int i)
{
    printf ("%d\n", i);
};
```

Also hat LLVM die Schleife nicht nur *unrolled* sondern auch die einfache Funktion `f()` *inlined* und den Körper der Schleife acht Mal generiert, anstatt die Schleife aufzurufen.

Dies ist möglich, wenn eine Funktion sehr einfach ist (wie meine) und wenn sie nicht allzu oft aufgerufen wird (wie hier).

ARM64: Optimierender GCC 4.9.1

Listing 1.148: Optimierender GCC 4.9.1

```
printing_function:
; zweites Argument von printf() vorbereiten:
```

```
        mov     w1, w0
; lade Adresse des "f(%d)\n" Strings
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; verzweige nur hier, anstatt Verzweigungen zu link und
return:
        b       printf
main:
; speichern von FP und LR auf dem lokalen Stack:
        stp     x29, x30, [sp, -32]!
; setze Stack Frame:
        add     x29, sp, 0
; Inhalt des X19 Registers auf dem Stack ablegen
        str     x19, [sp,16]
; wir verwenden das W19 Register als Zähler.
; weise diesem den Startwert 2 zu:
        mov     w19, 2
.L3:
; erstes Argument von printing_function() vorbereiten:
        mov     w0, w19
; Zählerregister inkrementieren.
        add     w19, w19, 1
; W0 enthält hier immer noch den Wert des Zählers vor dem Inkrement.
        bl      printing_function
; am Ende?
        cmp     w19, 10
; nein, dann springe zum Anfang des Schleifenkörpers:
        bne     .L3
; return 0
        mov     w0, 0
; wiederherstellen des X19 Registers:
        ldr     x19, [sp,16]
; wiederherstellen von FP und LR:
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "f(%d)\n"
```

ARM64: Nicht optimierender GCC 4.9.1

Listing 1.149: Nicht optimierender GCC 4.9.1 -fno-inline

```
.LC0:
        .string "f(%d)\n"
printing_function:
; speichere FP und LR auf dem lokalen Stack:
        stp     x29, x30, [sp, -32]!
; setze den Stack Frame:
        add     x29, sp, 0
; Inhalt des W0 Registers speichern:
        str     w0, [x29,28]
; lade Adresse des "f(%d)\n" Strings
```

```
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; Eingabewert vom Stack zurück ins W1 Register laden:
        ldr     w1, [x29,28]
; Aufruf von printf()
        bl      printf
; wiederherstellen von FP und LR:
        ldp     x29, x30, [sp], 32
        ret
main:
; speichere FP und LR auf dem lokalen Stack:
        stp     x29, x30, [sp, -32]!
; setze den Stack Frame:
        add     x29, sp, 0
; Zähler initialisieren
        mov     w0, 2
; auf dem Stack an zugewiesener Stelle ablegen:
        str     w0, [x29,28]
; Schleifenkörper überspringen und zur Bedingungsprüfung springen:
        b       .L3
.L4:
; lade Zähler nach W0.
; als erstes Argument der printing_function():
        ldr     w0, [x29,28]
; Aufruf printing_function():
        bl      printing_function
; Zähler erhöhen:
        ldr     w0, [x29,28]
        add     w0, w0, 1
        str     w0, [x29,28]
.L3:
; Bedingungsprüfung der Schleife.
; Zähler laden:
        ldr     w0, [x29,28]
; ist 9 erreicht?
        cmp     w0, 9
; kleiner oder gleich? dann springe zum Anfang der Schleife:
; sonst tue nichts.
        ble     .L4
; return 0
        mov     w0, 0
; wiederherstellen von FP und LR:
        ldp     x29, x30, [sp], 32
        ret
```

MIPS

Eine Sache noch

Im generierten Code sehen wir folgendes: nach der Initialisierung von i wird der Körper der Schleife nicht ausgeführt, da die Bedingung für i zuerst geprüft wird und erst danach der Körper der Schleife ausgeführt werden kann. Und so ist es auch

korrekt.

Denn, falls die Bedingung zu Beginn falsch ist, darf der Körper der Schleife nie ausgeführt werden. Dies ist z.B. im folgenden Fall möglich:

```
for (i=0; i<total_entries_to_process; i++)
    Schleifenkörper;
```

Wenn *total_entries_to_process* gleich 0 ist, darf der Körper der Schleife auf keinen Fall ausgeführt werden.

Deshalb wird die Bedingung stets vor der Ausführung geprüft.

Ein optimierter Compiler kann jedoch das Prüfen der Bedingung und den Schleifenkörper vertauschen, falls sichergestellt ist, dass die hier beschriebene Situation auf keinen Fall eintreten kann (wie in unserem sehr einfachen Beispiel und bei Keil, Xcode (LLVM), MSVC im optimierten Modus).

1.16.2 Funktion zum Kopieren von Speicherblöcken

Echte Funktionen zum Kopieren von Speicherblöcken kopieren in jedem Schritt 4 oder 8 Byte und verwenden SIMD[99], Vektorisierung, etc. Aber um einen Eindruck zu erhalten betrachte dieses einfachstmögliche Beispiel.

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Grundlegende Implementierung

Listing 1.150: GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
; RDI = Zieladresse
; RSI = Quelladresse
; RDX = Blockgröße

; initialisiere Zähler (i) mit 0
        xor     eax, eax
.L2:
; alles kopiert? dann verlassen:
        cmp     rax, rdx
        je      .L5
; load byte at RSI+i:
        mov     cl, BYTE PTR [rsi+rax]
; store byte at RDI+i:
```

99 Single Instruction, Multiple Data

```
        mov     BYTE PTR [rdi+rax], cl
        inc     rax ; i++
        jmp     .L2
.L5:
        ret
```

Listing 1.151: GCC 4.9 ARM64 optimized for size (-Os)

```
my_memcpy:
; X0 = Zieladresse
; X1 = Quelladresse
; X2 = Blockgröße

; initialisiere Zähler (i) mit 0
        mov     x3, 0
.L2:
; alles kopiert? dann verlassen:
        cmp     x3, x2
        beq     .L5
; load byte at X1+i:
        ldrb    w4, [x1,x3]
; store byte at X0+i:
        strb    w4, [x0,x3]
        add     x3, x3, 1 ; i++
        b       .L2
.L5:
        ret
```

Listing 1.152: Optimierender Keil 6/2013 (Thumb Modus)

```
my_memcpy PROC
; R0 = Zieladresse
; R1 = Quelladresse
; R2 = Blockgröße

        PUSH     {r4,lr}
; initialisiere Zähler (i) mit 0
        MOVS     r3,#0
; Bedingung wird am Ende der Schleife geprüft, daher springe dorthin:
        B        |L0.12|
|L0.6|
; lade Byte von R1+i:
        LDRB     r4,[r1,r3]
; speichere Byte an der Stelle R0+i:
        STRB     r4,[r0,r3]
; i++
        ADDS     r3,r3,#1
|L0.12|
; i<size?
        CMP      r3,r2
; springe zum Anfang der Schleife, falls es so ist:
        BCC      |L0.6|
        POP      {r4,pc}
        ENDP
```

ARM in ARM mode

Keil in ARM mode macht Gebrauch von konditionalen Suffixen:

Listing 1.153: Optimierender Keil 6/2013 (ARM Modus)

```
my_memcpy PROC
; R0 = Zieladresse
; R1 = Quelladresse
; R2 = Blockgröße

; initialisiere Zähler (i) mit 0
        MOV      r3,#0
|L0.4|
; alles kopiert?
        CMP      r3,r2
; der folgende Codeblock wird nur ausgeführt, falls die less than
; Bedingung, ; d.h., falls R2<R3 oder i<size wahr ist.
; lade Byte von R1+i:
        LDRBCC   r12,[r1,r3]
; speichere Byte an der Stelle R0+i:
        STRBCC   r12,[r0,r3]
; i++
        ADDCC    r3,r3,#1
; der letzte Befehl des conditional block.
; springe zum Anfang der Schleife, falls i<size
; andernfalls tue nichts (d.h., falls i>=size)
        BCC      |L0.4|
; return
        BX       lr
        ENDP
```

Deshalb gibt es hier nur einen Verzweigungsbefehl anstatt deren zwei.

MIPS

Listing 1.154: GCC 4.4.5 optimized for size (-Os) (IDA)

```
my_memcpy:
; springe zum Prüfen der Schleifenbedingung:
                b       loc_14
; initialisiere Zähler (i) mit 0
; bleibt stets in $v0:
                move    $v0, $zero ; branch delay slot

loc_8:                                  # CODE XREF: my_memcpy+1C
; lade unsigned Byte von $t0 nach $v1:
                lbu     $v1, 0($t0)
; erhöhe Zähler (i):
                addiu   $v0, 1
; speichere Byte an der Stelle $a3
                sb      $v1, 0($a3)

loc_14:                                 # CODE XREF: my_memcpy
```

```
; prüfe, ob Zähler (i) in $v0 immer noch kleiner ist als das dritte
; Funktionsargument ("cnt" in $a2):
                sltu    $v1, $v0, $a2
; Byteadresse in Quellblock bilden:
                addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; springe zum Schleifenkörper, falls der Zähler immer noch kleiner ist als
    "cnt":
                bnez    $v1, loc_8
; Byteadresse im Zielblock bilden ($a3 = $a0+$v0 = dst+i):
                addu    $a3, $a0, $v0 ; branch delay slot
; beenden, falls BNEZ nicht getriggert wurde:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

Hier tauchen zwei neue Befehle auf: LBU („Load Byte Unsigned“) und SB („Store Byte“).

Wie in ARM haben alle MIPS Register eine Breite von 32 bit, es gibt keine byte-breiten Teile wie bei x86.

Wenn wir also mit einzelnen Bytes arbeiten, müssen wir stets ein komplettes 32-bit-Register hierfür verwenden.

LBU lädt ein Byte und löscht alle anderen Bits („Unsigned“).

Der Befehl LB („Load Byte“) dagegen erweitert das geladene Byte zu einem vorzeichenbehafteten 32-bit-Wert.

SB schreibt ein Byte der niederwertigsten 8 Bit des Registers in den Speicher.

Vektorisierung

Optimierender GCC kann viel mehr; siehe dieses Beispiel: 1.27.1 on page 490.

1.16.3 Fazit

Gerüst einer Schleife von einschließlich 2 bis einschließlich 9:

Listing 1.155: x86

```
    mov [counter], 2 ; Initialisierung
    jmp check
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zählervariable auf lokalem Stack
    add [counter], 1 ; inkrementieren
check:
    cmp [counter], 9
    jle body
```

Das Inkrementieren kann in nicht optimiertem Code durch 3 Instruktionen dargestellt werden:

Listing 1.156: x86

```
    MOV [counter], 2 ; Initialisierung
    JMP check
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zählervariable auf lokalem Stack
    MOV REG, [counter] ; inkrementieren
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

Falls der Körper einer Schleife besonders kurz ist, kann ein Register als Zähler verwendet werden:

Listing 1.157: x86

```
    MOV EBX, 2 ; Initialisierung
    JMP check
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zähler in EBX, aber verändere ihn nicht!
    INC EBX ; inkrementieren
check:
    CMP EBX, 9
    JLE body
```

Einige Teile der Schleife können vom Compiler in unterschiedlichen Reihenfolgen generiert werden:

Listing 1.158: x86

```
    MOV [counter], 2 ; Initialisierung
    JMP label_check
label_increment:
    ADD [counter], 1 ; inkrementieren
label_check:
    CMP [counter], 10
    JGE exit
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zählervariable auf lokalem Stack
    JMP label_increment
exit:
```

Normalerweise wird die Bedingung *vor* dem Körper geprüft, aber der Compiler kann den Code auch so anordnen, dass die Bedingung *nach* dem Körper geprüft wird.

Dies geschieht dann, wenn der Compiler sicher sein kann, dass die Bedingung im ersten Durchlauf stets *wahr* ist, sodass der Körper der Schleife mindestens einmal tatsächlich ausgeführt wird:

Listing 1.159: x86

```
    MOV REG, 2 ; Initialisierung
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zähler in REG, aber verändere ihn nicht!
    INC REG ; inkrementieren
    CMP REG, 10
    JL body
```

Verwendung des *LOOP* Befehls. Sehr selten, Compiler verwenden ihn nicht. Wenn er auftaucht, ist dies ein Zeichen dafür, dass das entsprechende Codesegment von Hand geschrieben worden ist:

Listing 1.160: x86

```
    ; zähle von 10 bis 1
    MOV ECX, 10
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zähler in ECX, aber verändere ihn nicht!
    LOOP body
```

ARM.

Das R4 Register fungiert in diesem Beispiel als Zähler:

Listing 1.161: ARM

```
    MOV R4, 2 ; Initialisierung
    B check
body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zähler in R4, aber verändere ihn nicht!
    ADD R4,R4, #1 ; inkrementieren
check:
    CMP R4, #10
    BLT body
```

1.16.4 Übungen

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.17 Mehr über Zeichenketten

1.17.1 strlen()

German text placeholder

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

x86

Nicht optimierender MSVC

Kompilieren wir es:

```
_eos$ = -4                          ; size = 4
_str$ = 8                           ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; setze Pointer auf String von "str"
    mov     DWORD PTR _eos$[ebp], eax  ; setze ihn auf lokale Variable "eos"
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; nimm ein Byte von der Adresse in ECX und ;
    ; speichere es als 32-bit Wert mit Vorzeichen in EDX

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; erhöhe EAX
    mov     DWORD PTR _eos$[ebp], eax  ; setze EAX zurück auf "eos"
    test    edx, edx                   ; ist EDX null?
    je      SHORT $LN1@strlen_         ; ja, dann beende Schleife
    jmp     SHORT $LN2@strlen_         ; setze Schleife fort
$LN1@strlen_:

    ; hier berechnen wir die Differenz zwischen zwei Pointern.
```

```
    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                        ; subtrahiere 1 und gib Ergebnis
    zurück
    mov    esp, ebp pop    ebp
    ret    0
_strlen_ ENDP
```

Wir finden hier zwei neue Befehle: MOVSX und TEST.

Der erste –MOVSX–nimmt ein Byte aus einer Speicheradresse und speichert den Wert in einem 32-bit-Register. MOVSX steht für *MOV with Sign-Extend*. MOVSX setzt die übrigen Bits vom 8. bis zum 31. auf 1, falls das Quellbyte *negativ* ist oder auf 0, falls es *positiv* ist.

Und hier ist der Grund dafür.

Standardmäßig ist der *char* Datentyp in MSVC und GCC vorzeichenbehaftet (signed). Wenn wir zwei Werte haben, einen *char* und einen *int*, (*int* ist ebenfalls vorzeichenbehaftet) und der erste Wert enthält -2 (kodiert als 0xFE) und wir kopieren dieses Byte in den *int* Container, erhalten wir 0x000000FE und dies entspricht als signed *int* 254, aber nicht -2. Der signed *int*-2 wird als 0xFFFFFFFE dargestellt. Wenn wir also 0xFE vom Datentyp *char* nach *int* übertragen wollen, müssen wir das Vorzeichen identifizieren und den Wert entsprechend erweitern. Genau dies tut der Befehl MOVSX.

Es ist schwer zu sagen, ob der Compiler tatsächlich eine *char* Variable in EDX speichern muss, er könnte auch einen 8-Bit-Registerteil (z.B. DL) dafür verwenden . Offenbar arbeitet der Register Allokator des Compilers auf diese Art.

Wir finden im Weiteren den Befehl TEST EDX, EDX. Für mehr Informationen zum TEST Befehl siehe auch den Abschnitt über Bitfelder (1.21 on page 355). In unserem Fall überprüft der Befehl lediglich, ob der Wert im Register EDX gleich 0 ist.

Nicht optimierender GCC

Schauen wir uns GCC 4.4.1 an:

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
```

```
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

Das Ergebnis ist fast identisch mit dem von MSVC, aber hier finden wir MOVZX anstelle von MOVSX. MOVZX steht für *MOV with Zero-Extend*. Dieser Befehl kopiert einen 8-Bit- oder 16-Bit-Wert in ein 32-Bit-Register und setzt die übrigen Bits auf 0. Tatsächlich findet dieser Befehl vor allem deshalb Anwendung, weil er es uns erlaubt, folgendes Befehlspaar zu ersetzen:
`xor eax, eax / mov al, [...]`.

Andererseits ist offensichtlich, dass der Compiler folgenden Code erzeugen kann: `mov al, byte ptr [eax] / test al, al`–es ist fast das gleiche, aber die oberen Bits des EAX Registers enthalten hier Zufallswerte bzw. sogenanntes Zufallsrauschen. Aber bedenken wir den Nachteil des Compilers–er kann nicht leichter verständlichen Code erzeugen. Genau genommen, ist der Compiler überhaupt nicht daran gebunden, (Menschen) verständlichen Code zu erzeugen.

Der nächste neue Befehl für uns ist SETNZ. In diesem Fall setzt `test al,al` das ZF flag auf 0, falls AL nicht 0 enthät, aber SETNZ setzt AL auf 1, falls ZF==0 (ITNZ steht für *non zero*). In natürlicher Sprache, *falls AL ungleich 0, springe zu loc_80483F0*. Der Compiler erzeugt leicht redundanten Code, aber bedenken wir, dass die Optimierung hier deaktiviert ist.

Optimierender MSVC

Kompilieren wir nun alles in MSVC 2012 mit aktivierter Optimierung (/0x):

Listing 1.162: Optimierender MSVC 2012 /Ob0

```
_str$ = 8                    ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> Pointer auf den String
        mov     eax, edx                    ; verschiebe nach EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; nein, setze Schleife fort
        sub     eax, edx                    ; berechne Differenz der Pointer
        dec     eax                         ; dekrementiere EAX
```

```
        ret     0
_strlen ENDP
```

Jetzt ist alles einfacher. Unnötig zu erwähnen, dass der Compiler Register mit solcher Effizienz nur in kleinen Funktionen mit einigen wenigen lokalen Variablen verwenden kann.

INC/DEC—sind inkrement/dekrement Befehle; mit anderen Worten: addiere oder subtrahiere 1 zu bzw. von einer Variable.

Optimierender MSVC + OllyDbg

Wir untersuchen das (optimierte) Beispiel in OllyDbg. Hier ist der erste Durchlauf:

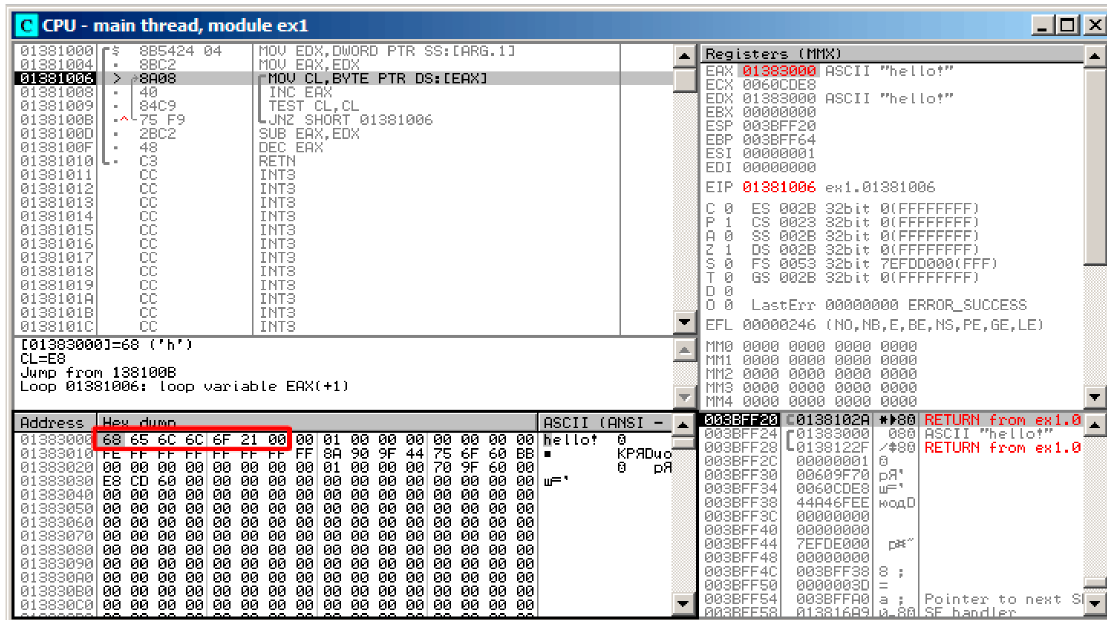


Abbildung 1.58: OllyDbg: Beginn erster Durchlauf

Wir sehen, dass OllyDbg eine Schleife gefunden hat, und zur Verbesserung der Lesbarkeit, diese in eckige Klammern *eingeschlossen* hat. Nach Rechtsklick auf EAX wählen wir „Follow in Dump“ und das Speicherfenster scrollt an die passende Stelle. Hier sehen wir den String „hello!“ im Speicher. Dahinter befindet sich mindestens ein Nullbyte und im Anschluss Zufallsbits.

Wenn OllyDbg ein Register mit einer gültigen Adresse, die auf einen String zeigt, findet, wird dieser String angezeigt.

Wir drücken einige Male F8 (step over) um zum Anfang der Schleifenkörpers zu gelangen:

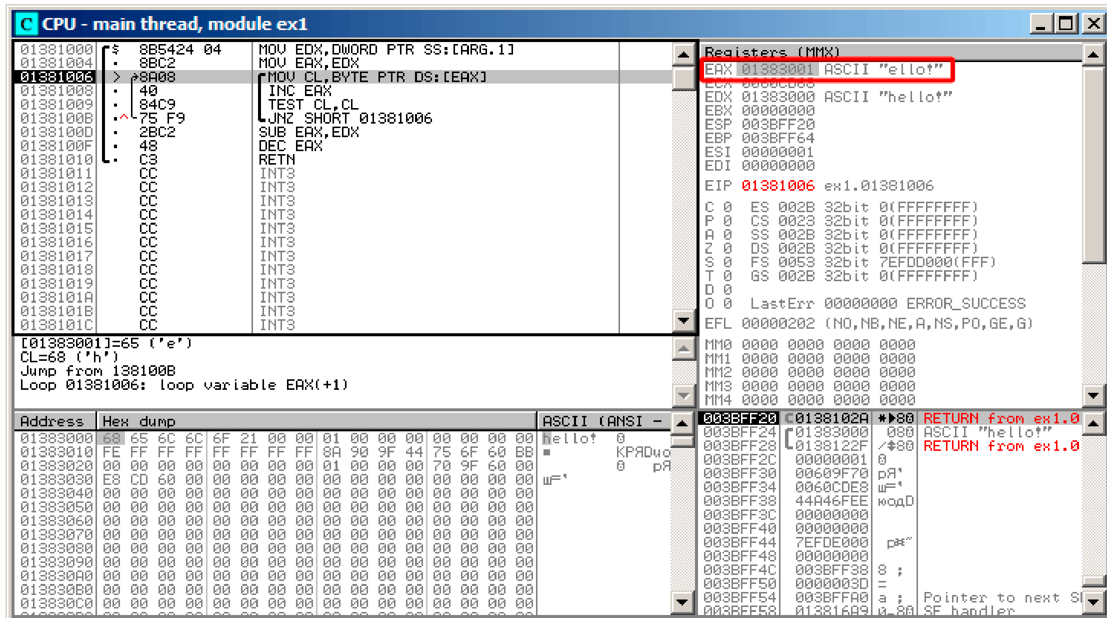


Abbildung 1.59: OllyDbg: Beginn zweiter Durchlauf

Wir sehen, dass EAX nun die Adresse des zweiten Zeichens des Strings enthält.

Durch hinreichend häufiges Drücken von F8 verlassen wir schließlich die Schleife:

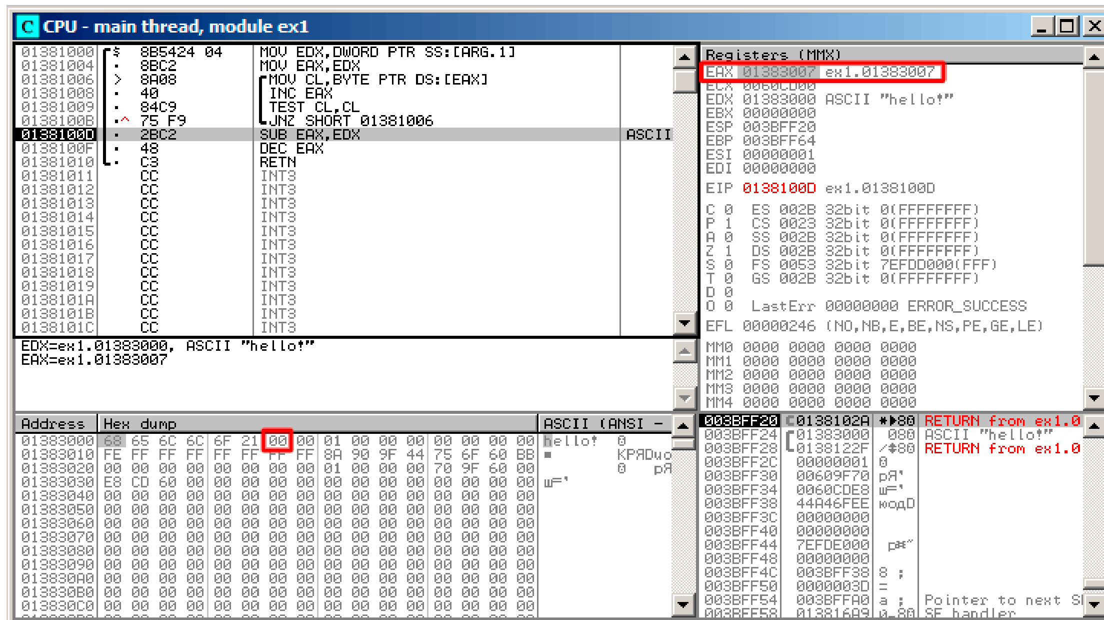


Abbildung 1.60: OllyDbg: Pointer Differenz wird berechnet

Wir sehen, dass EAX jetzt die Adresse des Nullbytes direkt hinter dem String enthält. In der Zwischenzeit hat sich EDX nicht verändert, es zeigt also immer noch auf den Anfang des Strings.

Die Differenz zwischen den beiden Adressen wird jetzt berechnet.

Der SUB Befehl wurde gerade ausgeführt:

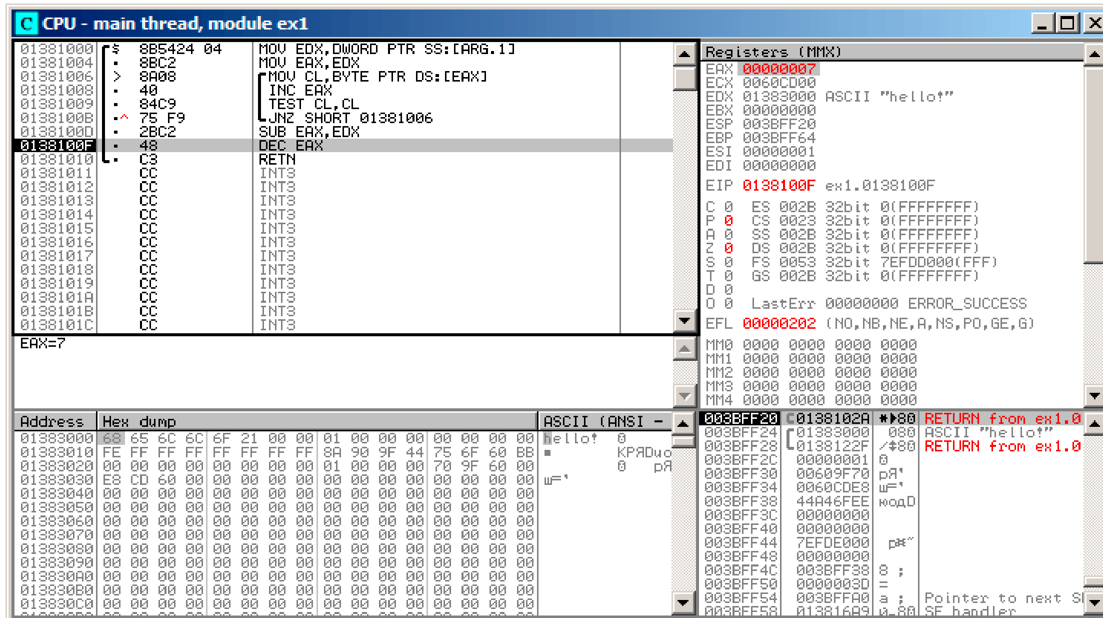


Abbildung 1.61: OllyDbg: EAX muss dekrementiert werden

Die Differenz der Pointer im EAX Register beträgt nun–7. Tatsächlich beträgt die Länge des „hello!“ Strings 6 Zeichen, aber mit dem Nullbyte am Ende dazugezählt sind es 7. Die Funktion `strlen()` soll aber die Anzahl der Nicht-Null-Zeichen im String zurückliefert, also wird einmal dekrementiert und der Funktionsaufruf anschließend beendet.

Optimierender GCC

Schauen wir uns GCC 4.4.1 mit aktiverter Optimierung (-O3 key) an:

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
```

```
                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

Hier erzeugt GCC fast identischen Code zu MSVC, außer dass hier ein MOVZX auftritt. In der Tat könnte MOVZX hier durch mov dl, byte ptr [eax] ersetzt werden.

Möglicherweise ist es einfacher für den GCC Code Generator sich daran zu *erinnern*, dass das gesamte 32-bit-EDX Register für eine *char* Variable reserviert ist und so sicherzustellen, dass die oberen Bits zu keinem Zeitpunkt Zufallsrauschen enthalten.

Danach finden wir also einen neuen Befehl–NOT. Dieser Befehl kippt alle Bits in seinem Operanden.
Man kann sagen, dass es sich um ein Synonym zum Befehl XOR ECX, 0ffffffffh handelt. NOT und das darauf folgende ADD berechnen die Differenz im Pointer und subtrahieren 1, nur auf eine andere Art und Weise. Zu Beginn wird ECX, in dem der Pointer auf *str* gespeichert ist, invertiert und vom Ergebnis wird 1 abgezogen.

Mit anderen Worten, am Ende der Funktion, direkt nach dem Schleifenkörper, werden die folgenden Befehle ausgeführt:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

… und das ist äquivalent zu:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Warum GCC entschieden hat, dass das eine besser ist als das andere? Schwer zu sagen. Möglicherweise sind aber beide Variante gleichermaßen effizient.

ARM

32-bit ARM

Nicht optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Listing 1.163: Nicht optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
_strlen
```

```
eos  = -8
str  = -4

     SUB    SP, SP, #8 ; reserviere 8 Bytes für lokale Variablen
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; setze 8 Bytes an Speicher frei
     BX     LR
```

Der nicht optimierende LLVM erzeugt zu viel Code, aber wir können wir erkennen wir die Funktion mit lokalen Variablen auf dem Stack arbeitet. Es gibt nur zwei lokale Variablen in unserer Funktion: *eos* und *str*. In folgenden von IDA erzeugten Listing, sind die Variablen *var_8* und *var_4* in *eos* bzw. *str* umbenannt.

Der erste Befehl speichert lediglich bei Eingabewerte in *str* und *eos*.

Der Körper der Schleife startet beim Label *loc_2CB8*.

Die ersten drei Befehle des Schleifenkörpers (LDR, ADD, STR) laden den Wert von *eos* nach R0. Anschließend wird der Wert erhöht und zurück in *eos* auf den Stack geschrieben.

Der folgende Befehl, LDRSB R0, [R0] („Load Register Signed Byte“), lädt ein Byte aus dem Speicher von der Adresse in R0 und erweitert es mit Vorzeichen auf 32-bit. [100]. Dies ist vergleichbar zum MOVSX Befehl in x86.

Der Compiler behandelt dieses Byte als signed, das der *char* Typ nach dem C-Standard ebenfalls signed ist. Dies wurde in Bezug auf x86 in diesem Abschnitt bereits in (1.17.1 on page 233) beschrieben.

Man beachte, dass es in ARM unmöglich ist, einen 8- oder 16-bit-Teil eines 32-bit-Registers alleine zu verwenden, anderes als in x86.

Dies rührt daher, dass x86 eine große Bandbreite and Kompatibilität mit Vorgängerversionen besitzt, bis hin zum 16-bit 8086 oder sogar dem 8-bit 8080, ARM auf der anderen Seite jedoch von Beginn an als 32-bit RISC-Prozessor geplant wurde.

[100] Der Keil Compiler behandelt den Typ *char* als signed, genau wie MSVC und GCC.

Infolgedessen müssen auch um einzelne Bytes in ARM zu verarbeiten, stets komplette 32-bit-Register verwendet werden.

Der Befehl LDRSB lädt nun die Bytes des String einzeln nach R0. Die nachfolgenden CMP und BEQ Befehle prüfen, ob das aktuelle Byte 0 ist. Wenn nicht, beginnt der Körper der Schleife erneut. Und wenn das aktuelle Byte 0 ist, dann wird die Schleife beendet.

Am Ende der Funktion wird die Differenz zwischen *eos* und *str* berechnet, 1 vom Ergebnis abgezogen und das Resultat über das Register R0 zurückgegeben.

N.B. Register wurden in dieser Funktion nicht gespeichert. Das liegt daran, dass gemäß der ARM Aufrufkonventionen die Register R0 bis R3 sogenannte „scratch register" sind, vorgesehen für Parameterübergaben. Deshalb ist es nicht notwendig ihren Inhalt am Ende der Funktion wiederherzustellen, denn die aufrufende Funktion wird diese Werte nicht weiter verwenden. Im weiteren können diese für alles Mögliche benutzt werden.

Es werden hier keine weiteren Register verwendet, sodass wir nichts auf dem Stack speichern müssen.

Dadurch kann der control flow über einen einfachen Sprung (BX) an die aufrufende Funktion an der Adresse im LR Register übergeben werden.

Optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

Listing 1.164: Optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

```
_strlen
           MOV      R1, R0

loc_2DF6
           LDRB.W   R2, [R1],#1
           CMP      R2, #0
           BNE      loc_2DF6
           MVNS     R0, R0
           ADD      R0, R1
           BX       LR
```

Der optimierende LLVM entscheidet also, dass *eos* und *str* keinen Platz auf dem Stack benötigen, sondern stets in Registern gespeichert werden können.

Vor dem Anfang des Schleifenkörpers befindet sich *str* stets in R0 und *eos* in R1.

Der Befehl LDRB.W R2, [R1],#1 lädt ein Byte aus dem Speicher von der Adresse aus R1 nach R2, erweitert es zum einem signed 32-bit-Wert und mehr noch: Das #1 am Ende des Befehl bewirkt „Post-indexed addressing", was bedeutet, dass 1 zum Register R1 addiert wird, nachdem das Byte geladen wurde. Mehr zum Thema: 1.30.2 on page 524.

Des Weiteren finden wir CMP und BNE[101] im Körper der Schleife; diese Befehle werden durchlaufen, bis 0 im String gefunden wurde.

[101](PowerPC, ARM) Branch if Not Equal

MVNS[102] (invertiert alle Bits wie NOT in x86) und ADD Befehle berechnen $eos - str - 1$. Tatsächlich berechnen diese beiden Befehle $R0 = str + eos$, was äquivalent zur Formulierung im Quellcode ist und die Begründung dazu wurde bereits hier gegeben (1.17.1 on page 240).

Offenbar befindet LLVM genau wie GCC, dass diese Code kürzer (oder schneller) ist.

Optimierender Keil 6/2013 (ARM Modus)

Listing 1.165: Optimierender Keil 6/2013 (ARM Modus)

```
_strlen
                MOV     R1, R0

loc_2C8
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

Fast das gleiche wie zuvor, mit der Änderung, dass der $str - eos - 1$ Ausdruck nicht am Ende der Funktion, sondern mitten in der Schleife berechnet wird. Wir erinnern uns, dass der -EQ Suffix bedeutet, dass die dieser Befehl nur dann ausgeführt wird, wenn die Operanden im CMP direkt davor gleich waren. Dadurch werden beide SUBEQ Befehle ausgeführt, falls das R0 Register 0 enthält und das Ergebnis verbleibt in R0.

ARM64

Optimierender GCC (Linaro) 4.9

```
my_strlen:
        mov     x1, x0
        ; X1 ist der temporäre Pointer (eos), verhält sich wie ein Cursor
.L58:
        ; lade Byte von X1 nach W2, erhöhe X1 (post-index)
        ldrb    w2, [x1],1
        ; Vergleich und Verzweigung, falls nicht null: vergleiche W2 mit 0,
        ; springe nach .L58, falls ungleich
        cbnz    w2, .L58
        ; berechne Differenz zwischen ursprünglichem Pointer in X0 und
        ; aktueller Adresse in X1
        sub     x0, x1, x0
        ; dekrementiere niedere 32-bit des Ergebnisses
        sub     w0, w0, #1
        ret
```

[102] MoVe Not

Der Algorithmus ist der gleiche wie in 1.17.1 on page 234: finde ein Nullbyte, berechne die Differenz zwischen den Pointern und subtrahiere 1 vom Ergebnis. Einige Kommentare wurden vom Autor hinzugefügt.

Die einzig bemerkenswerte Sache ist, dass unser Beispiel in gewisser Weise fehlerhaft ist:
`my_strlen()` liefert einen 32-bit *int*, obwohl es `size_t` oder einen anderen 64-bit Typ zurückliefern müsste.

Der Grund dafür ist, dass `strlen()` theoretisch für einen sehr großen Speicherblock, größer als 4GB, aufgerufen werden könnte und deshalb auf einer 64-bit-Plattform in der Lage sein muss, einen 64-bit-Wert zurückzuliefern.

Aufgrund meines Fehlers, arbeitet der letzte SUB Befehl nur mit einem 32-bit-Teil des Registers, wohingegen der vorletzte SUB Befehl mit dem kompletten 64-bit-Register arbeitet (und die Differenz zwischen den Pointer berechnet).

Es handelt sich um einen Fehler von mir, und es ist besser es so zu lassen, als ein Lehrbeispiel wie Code in einem derartigen Fall aussehen kann.

Nicht optimierender GCC (Linaro) 4.9

```
my_strlen:
; Funktionsprolog
        sub     sp, sp, #32
; erstes Argument (str) wird in [sp,8] gespeichert
        str     x0, [sp,8]
        ldr     x0, [sp,8]
; kopiere "str" in die "eos" Variable
        str     x0, [sp,24]
        nop
.L62:
; eos++
        ldr     x0, [sp,24]  ; lade "eos" nach X0
        add     x1, x0, 1    ; erhöhe X0
        str     x1, [sp,24]  ; sichere X0 in "eos"
; lade Byte aus dem Speicher von Adresse in X0 nach W0
        ldrb    w0, [x0]
; null gefunden? (das 32-bit Register WZR enthält stets 0)
        cmp     w0, wzr
; springe, falls nicht null (Zweig ungleich)
        bne     .L62
; Nullbyte gefunden, berechne jetzt Differenz
; lade "eos" nach X1
        ldr     x1, [sp,24]
; lade "str" nach X0
        ldr     x0, [sp,8]
; berechne Differenz
        sub     x0, x1, x0
; dekrementiere Ergebnis
        sub     w0, w0, #1
; Funktionsepilog
        add     sp, sp, 32
```

```
        ret
```

Es ist umfangreicher. Die Variablen werden hier viel im Speicher (lokaler Stack) herumgeschoben. Der obige Fehler findet sich auch hier: das Dekrementieren geschieht nur in einem 32-bit-Teil des Registers.

MIPS

Listing 1.166: Optimierender GCC 4.4.5 (IDA)

```
my_strlen:
; "eos" Variable bleibt stets in $v1:
                move    $v1, $a0

loc_4:
; lade Byte von der Adresse in "eos" nach $a1:
                lb      $a1, 0($v1)
                or      $at, $zero ; lade delay slot, NOP
; ist das geladene Byte ungleich 0, springe nach loc_4:
                bnez    $a1, loc_4
; erhöhe "eos" in jedem Falle:
                addiu   $v1, 1 ; branch delay slot
; Schleife beendet. Invertiere "str" Variable:
                nor     $v0, $zero, $a0
; $v0=-str-1
                jr      $ra
; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
                addu    $v0, $v1, $v0 ; branch delay slot
```

MIPS besitzt keinen NOT Befehl, dafür aber den Befehl NOT, welcher der Funktion OR + NOT entspricht.

Diese Funktion wird häufig in der Digitaltechnik verwendet[103].

Der Apollo Guidance Computer, der im Apollo Programm der NASA verwendet wurde, bestand beispielsweise ausschließlich aus 5600 NOR Gattern: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. In der Programmierung ist die Funktion NOT nicht besonders beliebt.

Die NOT Funktion ist hier also durch NOR DST, $ZERO, SRC implementiert.

Aus dem Grundlagenteil wissen wir, dass das bitweise invertieren einer vorzeichenbehafteten Zahl gerade einem Wechsel des Vorzeichens mit anschließender Subtraktion von 1 entspricht.

Was NOT hier also tut, ist, den Wert von str in $-str - 1$ umzuwandeln. Die folgende Addition bereitet das Ergebnis vor.

[103] NOR wird „universelles Gatter“ genannt

1.18 Ersetzen von arithmetischen Operationen

Beim Optimierungsvorgang kann eine Instruktion durch eine andere ersetzt werden, oder sogar durch eine Instruktionsgruppe. So können beispielsweise ADD und SUB einander ersetztten, siehe Zeile 18 in Listing.**??**.

Die LEA Instruktion wird z.B. oft verwendet um einfache arithmetische Berechnungen durchzuführen, siehe **??** on page ??.

1.18.1 Multiplikation

Multiplikation durch Addition

Hier ist ein einfaches Beispiel:

```
unsigned int f(unsigned int a)
{
        return a*8;
};
```

Die Multiplikation mit 8 wird durch 3 Additionsbefehle ersetzt, welche das gleiche Ergebnis erzielen. Offenbar hat der MSVC Optimierer entschieden, dass der Code so schneller sein kann.

Listing 1.167: Optimierender MSVC 2010

```
_TEXT   SEGMENT
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, eax
        add     eax, eax
        add     eax, eax
        ret     0
_f      ENDP
_TEXT   ENDS
END
```

Multiplikation durch Verschieben

Multiplikation mit und Divisionen durch Zahlen, die Potenzen von 2 sind, werden oft durch Schiebebefehle (oft auch Shifting genannt) ersetzt.

```
unsigned int f(unsigned int a)
{
        return a*4;
};
```

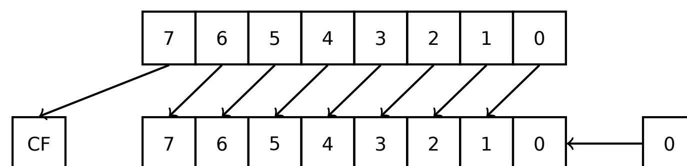
Listing 1.168: Nicht optimierender MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
```

```
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        shl     eax, 2
        pop     ebp
        ret     0
_f      ENDP
```

Die Multiplikation mit 4 entspricht einer Linksverschiebung der Zahl um 2 Bit und Einfügen zweier Nullen an der rechten Seite (an den niederwertigsten beiden Bits). Das Prinzip ist das gleiche wie bei der dezimalen Multiplikation von 3 mit 100 –wir schreiben einfach zwei Nullen rechts an die Zahl.

Der Befehl für Linksverschiebung funktioniert wie folgt:



Die beiden rechts angefügten Bits sind stets Nullen.

Multiplikation mit 4 in ARM:

Listing 1.169: Nicht optimierender Keil 6/2013 (ARM Modus)

```
f PROC
        LSL      r0,r0,#2
        BX       lr
        ENDP
```

Multiplikation mit 4 in MIPS:

Listing 1.170: Optimierender GCC 4.4.5 (IDA)

```
      jr      $ra
      sll     $v0, $a0, 2 ; branch delay slot
```

SLL bedeutet „Shift Left Logical“.

Multiplikation durch Verschieben, Subtrahieren und Addieren

Es ist auch möglich die Multiplikation zu ersetzen, wenn man mit Zahlen wie 7 oder 17 multipliziert, wenn Verschiebung verwendet wird. Die zugrundeliegende Mathematik ist relativ einfach.

32-bit

```
#include <stdint.h>

int f1(int a)
```

```
{
        return a*7;
};

int f2(int a)
{
        return a*28;
};

int f3(int a)
{
        return a*17;
};
```

x86

Listing 1.171: Optimierender MSVC 2012

```
; a*7
_a$ = 8
_f1     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        ret     0
_f1     ENDP

; a*28
_a$ = 8
_f2     PROC
        mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
        lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
        sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
        shl     eax, 2
; EAX=EAX<<2=(a*7)*4=a*28
        ret     0
_f2     ENDP

; a*17
_a$ = 8
_f3     PROC
        mov     eax, DWORD PTR _a$[esp-4]
; EAX=a
        shl     eax, 4
; EAX=EAX<<4=EAX*16=a*16
        add     eax, DWORD PTR _a$[esp-4]
```

```
; EAX=EAX+a=a*16+a=a*17
        ret     0
_f3     ENDP
```

ARM

Keil im ARM mode benutzt den Umwandler zur Verschiebung im zweiten Operanden:

Listing 1.172: Optimierender Keil 6/2013 (ARM Modus)

```
; a*7
||f1|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        BX       lr
        ENDP

; a*28
||f2|| PROC
        RSB      r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
        LSL      r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
        BX       lr
        ENDP

; a*17
||f3|| PROC
        ADD      r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
        BX       lr
        ENDP
```

Da es im Thumb mode keine solchen Umwandler gibt, kann folglich `f2()` nicht optimiert werden:

Listing 1.173: Optimierender Keil 6/2013 (Thumb Modus)

```
; a*7
||f1|| PROC
        LSLS     r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS     r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX       lr
        ENDP

; a*28
||f2|| PROC
        MOVS     r1,#0x1c ; 28
; R1=28
        MULS     r0,r1,r0
```

```
; R0=R1*R0=28*a
        BX      lr
        ENDP

; a*17
||f3|| PROC
        LSLS    r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS    r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX      lr
        ENDP
```

MIPS

Listing 1.174: Optimierender GCC 4.4.5 (IDA)

```
_f1:
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
                sll     $a0, 2
; $a0 = $a0<<2 = $a0*4
                jr      $ra
                subu    $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
                jr      $ra
                addu    $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
        return a*7;
};

int64_t f2(int64_t a)
```

```
{
        return a*28;
};

int64_t f3(int64_t a)
{
        return a*17;
};
```

x64

Listing 1.175: Optimierender MSVC 2012

```
; a*7
f1:
        lea     rax, [0+rdi*8]
; RAX=RDI*8=a*8
        sub     rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
        ret

; a*28
f2:
        lea     rax, [0+rdi*4]
; RAX=RDI*4=a*4
        sal     rdi, 5
; RDI=RDI<<5=RDI*32=a*32
        sub     rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
        mov     rax, rdi
        ret

; a*17
f3:
        mov     rax, rdi
        sal     rax, 4
; RAX=RAX<<4=a*16
        add     rax, rdi
; RAX=a*16+a=a*17
        ret
```

ARM64

GCC 4.9 für ARM64 fasst sich dank der Verschiebe-Umwandler ebenfalls kurz:

Listing 1.176: Optimierender GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
        lsl     x1, x0, 3
```

```
; X1=X0<<3=X0*8=a*8
        sub     x0, x1, x0
; X0=X1-X0=a*8-a=a*7
        ret

; a*28
f2:
        lsl     x1, x0, 5
; X1=X0<<5=a*32
        sub     x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
        ret

; a*17
f3:
        add     x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
        ret
```

Booths Multiplikationsalgorithmus

Es gab Zeiten, in denen Computer groß und so teuer waren, dass einige von ihnen keinen Hardwaresupport für die Multiplikation in der CPU besaßen, so zum Beispiel der Data General Nova. Wenn dort eine Multiplikation benötigt wurde, musste diese softwareseitig abgebildet werden, zum Beispiel durch Booths Multiplikationsalgorithmus. Dabei handelt es sich um einen Algorithmus zur Multiplikation, welcher lediglich Addtionen und Verschiebeoperationen verwendet.

Zwar gehen moderne optimierende Compiler hier anders vor, aber das Ziel (die Multiplikation) und die Ressourcenfrage (schnellere Operationen) sind gleich.

1.18.2 Division

Division durch Verschieben

Beispiel der Division durch 4:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

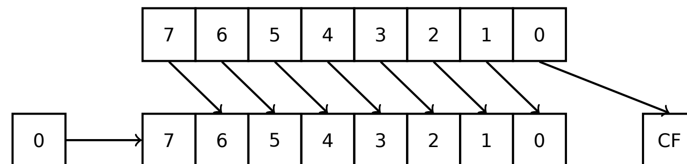
Wir betrachten (MSVC 2010):

Listing 1.177: MSVC 2010

```
_a$ = 8         ; size = 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

Der Befehl SHR (*Shift Right*) verschiebt die Zahl in diesem Beispiel um 2 Bits nach rechts. Die beiden freien Bits am linken Rand (dies sind die beiden höchstwertigsten Bits) werden auf null gesetzt. Die beiden niederwertigsten Bits werden entfernt. Diese beiden entfernten Bits entsprechen genau dem Rest der Division.

Der SHR Befehl funktioniert genau wie SHL, aber in die entgegengesetzte Richtung.



Das Vorgehen kann leicht verdeutlicht werden, wenn wir es an der Zahl 23 im Dezimalsystem veranschaulichen. Die 23 kann einfach durch 10 geteilt werden, indem die letzte Ziffer (3–Rest der Division) entfernt wird. Die 2 bleibt bei der Division als ganzzahliger Quotient übrig.

Der Rest wird also entfernt, was aber kein Problem darstellt, da wir hier ausschließlich mit ganzzahligen Werten arbeiten und nicht mit reellen Zahlen.

Division durch 4 in ARM:

Listing 1.178: Nicht optimierender Keil 6/2013 (ARM Modus)

```
f PROC
        LSR      r0,r0,#2
        BX       lr
        ENDP
```

Division by 4 in MIPS:

Listing 1.179: Optimierender GCC 4.4.5 (IDA)

```
        jr      $ra
        srl     $v0, $a0, 2 ; branch delay slot
```

Der Befehl SRL steht für „Shift Right Logical".

1.18.3 Übung

- http://challenges.re/59

1.19 Gleitkommaeinheit

Die FPU is ein Gerät innerhalb der CPU, welche speziell für den Umgang mit Fließkommazahlen ausgelegt ist.

In der Vergangenheit wurde die FPU auch als „Koprozessor" bezeichnet und sie befindet sich neben der CPU.

1.19.1 IEEE 754

Eine Zahl besteht gemäß IEEE 754 Format aus einem *Vorzeichne*, einer *Mantisse* (auch *Bruch* genannt) und einem *Exponenten*.

1.19.2 x86

Es lohnt sich einen Blick auf die Stackmaschine zu werfen oder die Grundlagen der Sprache Forth zu erlernen, bevor man die FPU in x86 genauer untersucht.

Es ist interessant zu wissen, dass sich der Koprozessor in der Vergangenheit (vor dem 80486 Chip) auf einem separaten Chip befand und nicht immer auf dem Mainboard vorinstalliert war. Es war möglich, diesen separat zu kaufen und zu installieren[104].

Seit der 80486 DX CPU ist die FPU in die CPU integriert.

Der Befehl FWAIT erinnert uns an diese Tatsache–er lässt die CPU in einen Wartezustand wechseln, in dem sie verbleibt, bis die FPU ihre Arbeit beendet hat.

Ein anderes Überbleibsel ist die Tatsache, dass die Opcodes der FPU Befehle mit sogenannten „escape"-Opcodes (D8..DF, d.h. Opcodes, die an einen separaten Koprozessor übergeben werden) beginnen.

Die FPU besitzt einen Stack, auf dem 8 80-bit-Register Platz finden, wobei jedes Register eine Zahl im IEEE 754 Format aufnehmen kann.

Es gibt ST(0)..ST(7). Verkürzend stellen IDA und OllyDbg ST(0), welches in manchen (Hand-)büchern als „Stack Top" dargestellt wird, als ST dar.

1.19.3 ARM, MIPS, x86/x64 SIMD

In ARM und MIPS ist die FPU kein Stack, sondern ein Registersatz.

Das gleiche Paradigma wird in den SIMD-Erweiterungen von x86/x64 CPUs verwendet.

1.19.4 C/C++

Die Standardsparche C/C++ bietet zumindest two unterschiedliche Fließkommezahlentypen, *float* (*einfache Genauigkeit*, 32 Bit) [105] und *double* (*doppelte Genauigkeit*, 64 Bit).

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997)246] können wir nachlesen, dass *einfache Genauigkeit* bedeutet, dass die Fließkommazahl in ein einzelnes (32-Bit)-Wort hineinpasst, *doppelte Genauigkeit* bedeutet dagegen, dass die Zahl in zwei Worten (64 Bit) abgelegt werden kann.

[104] John Carmack z.B. verwendete Fixpunktarithmetikwerte seinem Videospiel Doom, gespeichert in 32-bit GPR Registern (16 Bit für den Hauptteil und weitere 16 Bit für den gebrochenen Teil), sodass Doom auf 32-Bit-Computern ohne FPU, d.h., 80386 und 80486 SX, lauffähig war.

[105] Fließkommazahlen mit einfacher Genauigkeit werden auch im Abschnitt *Gleitkomma Datentypen als Struktur behandeln* (1.23.6 on page 442) behandelt

GCC unterstützt im Gegensatz zu MSVC ebenfalls den *long double* Typ (*erweiterte Genauigkeit* 80 Bit).

Der *float* Typ benötigt dieselbe Anzahl an Bits wie der *int* Typ in 32-Bit-Umgebungen, aber die Darstellung der Zahlen ist komplett verschieden voneinander.

1.19.5 Einfaches Beispiel

Betrachten wir folgendes einfache Beispiel:

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

x86

MSVC

Kompilieren mit MSVC 2010 liefert:

Listing 1.180: MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f  PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _a$[ebp]

; aktueller Stand des Stacks: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; aktueller Stand des Stacks: ST(0) = Ergebnis von _a geteilt durch 3.14

    fld    QWORD PTR _b$[ebp]
```

```
; aktueller Stand des Stacks: ST(0) = _b;
; ST(1) = Ergebnis von _a geteilt durch 3.14

    fmul   QWORD PTR __real@4010666666666666

; aktueller Stand des Stacks:
; ST(0) = Ergebnis von _b * 4.1;
; ST(1) = Ergebnis von _a geteilt durch 3.14

    faddp  ST(1), ST(0)

; aktueller Stand des Stacks: ST(0) = Ergebnis der Addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD nimmt 8 Byte vom Stack und lädt die Zahl in das ST(0) Register, wobei diese automatisch in das interne 80-bit-Format (*erweiterte Genauigkeit*) konvertiert wird.

FDIV teilt den Wert in ST(0) durch die Zahl, die an der Adresse __real@40091eb851eb851f gespeichert ist —der Wert 3.14 ist hier kodiert. Die Syntax des Assemblers erlaubt keine Fließkommazahlen, sodass wir hier die hexadezimale Darstellung von 3.14 im 64-bit IEEE 754 Format finden.

Nach der Ausführung von FDIV enthält ST(0) den Quotienten.

Es gibt übrigens auch noch den FDIVP Befehl, welcher ST(1) durch ST(0) teilt, beide Werte vom Stack holt und das Ergebnis ebenfalls auf dem Stack ablegt. Wer mit der Sprache Forth vertraut ist, erkennt schnell, dass es sich hier um eine Stackmaschine handelt.

Der nachfolgende FLD Befehl speichert den Wert von b auf dem Stack.

Anschließend wir der Quotient in ST(1) abgelegt und ST(0) enthält den Wert von b.

Der nächste FMUL Befehl führt folgende Multiplikation aus: b aus Register ST(0) wird mit dem Wert an der Speicherstelle __real@4010666666666666 (hier befindet sich die Zahl 4.1) multipliziert und hinterlässt das Ergebnis im ST(ß) Register.

Der letzte FADDP Befehl addiert die beiden Werte, die auf dem Stack zuoberst liegen, speichet das Ergebnis in ST(1) und holt dann den Wert von ST(0) vom Stack, wobei das oberste Element auf dem Stack in ST(0) gespeichert wird.

Die Funktion muss ihr Ergebnis im ST(0) Register zurückgeben, sodass außer dem Funktionsepilog nach FADDP keine weiteren Befehle mehr folgen.

MSVC + OllyDbg

Zwei Paare aus 32-bit Worten sind im Stack rot markiert. Jedes Paar ist eine double-Zahl im IEEE 754 Format und wurde von `main()` übergeben.

Wie sehen wie zunächst `FLD` einen Wert (1.2) von Stack lädt und diesen in `ST(0)` ablegt:

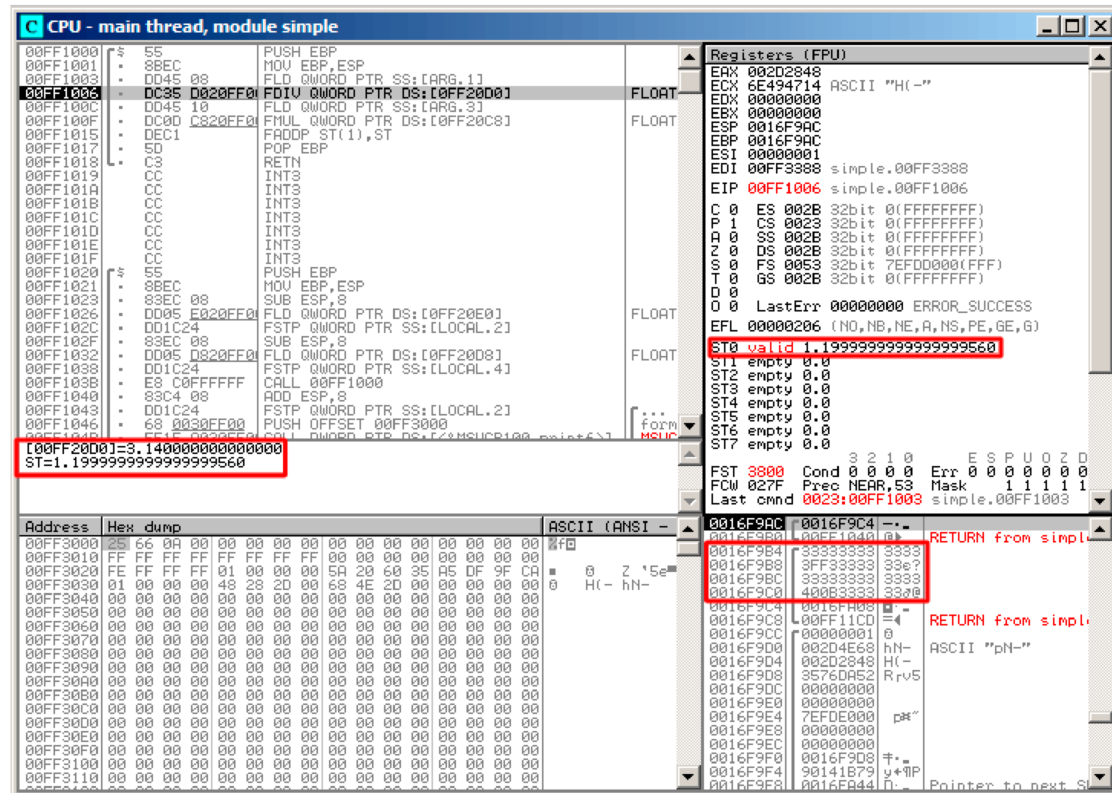


Abbildung 1.62: OllyDbg: der erste FLD wurde ausgeführt

Aufgrund der unvermeidlichen Konversionsfehler von der 64-bit IEEE 754 Fließkommazahl in ein 80-bit-Format (das intern in der FPU verwendet wird), sehen wird hier 1.1999…, was näherungsweise 1.2 entspricht.

EIP zeigt nun auf den nächsten Befehl (FDIV), der eine double-Zahl (eine Konstante) aus dem Speicher lädt. Zur besseren Übersicht zeigt OllyDbg deren Wert an: 3.14

Verfolgen wir das ganze etwas weiter. FDIV wurde ausgeführt, nun enthält ST(0) also 0.382…(Quotient):

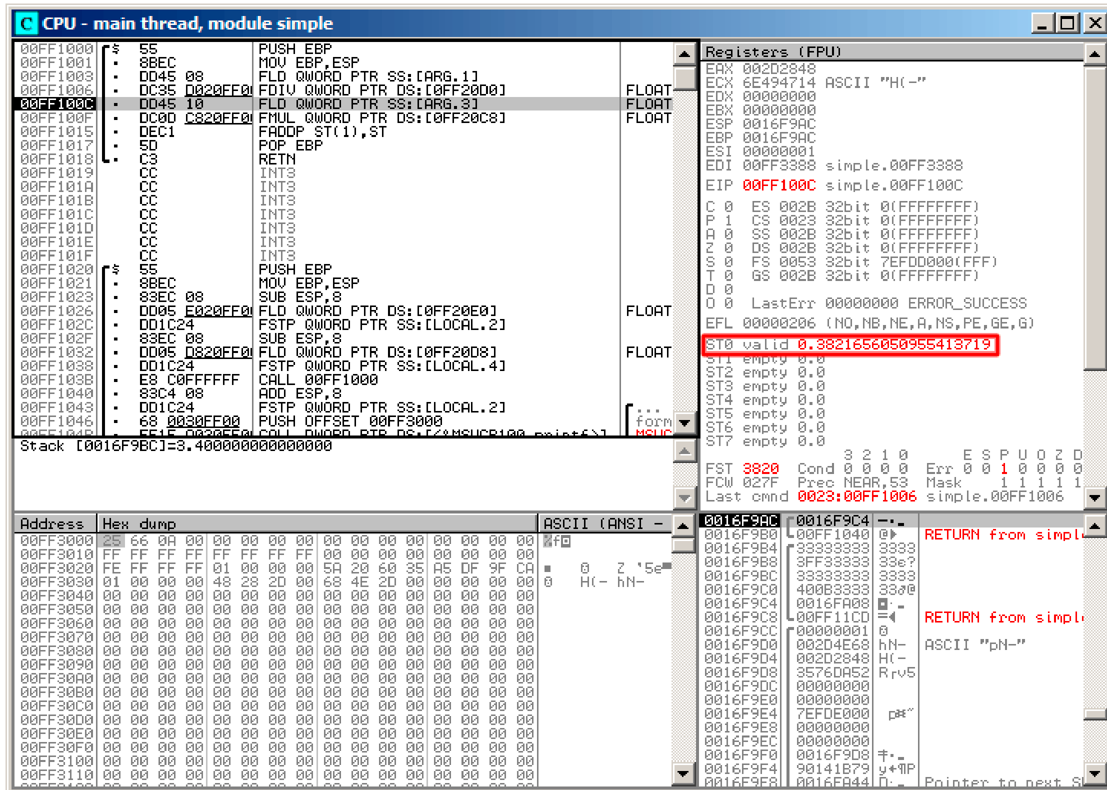


Abbildung 1.63: OllyDbg: FDIV wurde ausgeführt

Dritter Schritt: der nächste FLD Befehl wurde ausgeführt; er lud 3.4 nach ST(0) (wir sehen hier den Näherungswert 3.39999...):

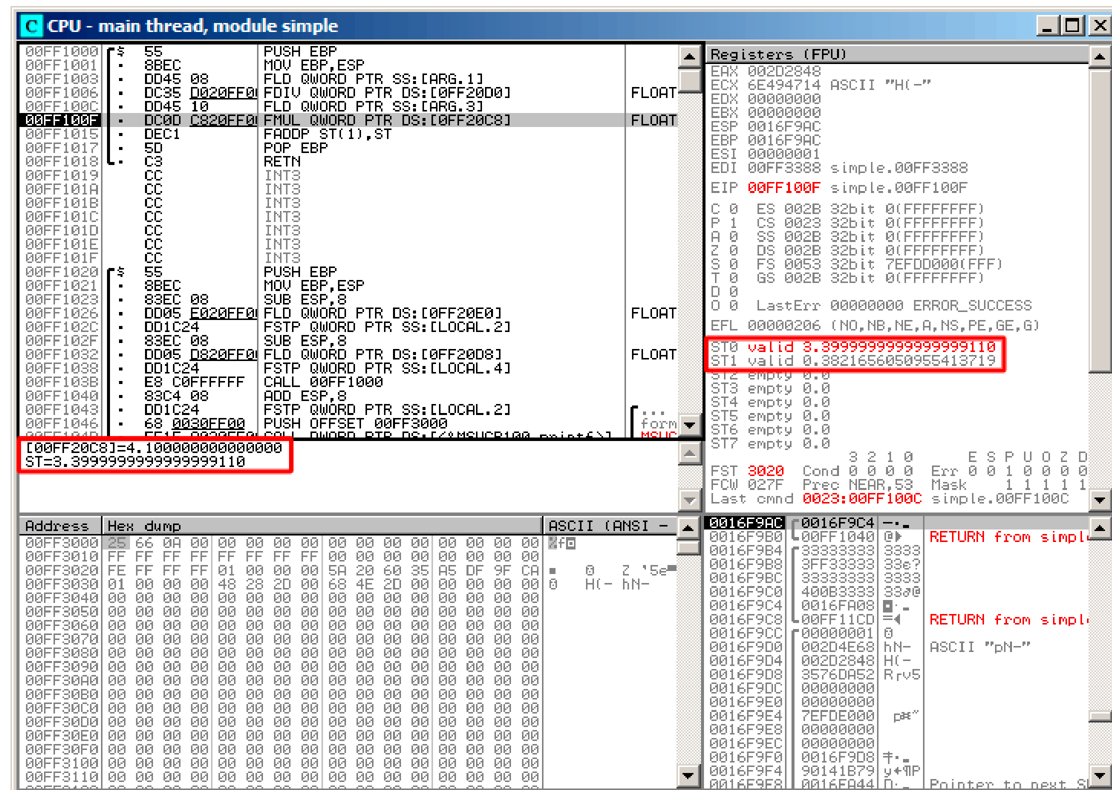


Abbildung 1.64: OllyDbg: der zweite FLD wurde ausgeführt

Gleichzeitig wird der Quotient nach ST(1) verschoben. In diesem Moment zeigt EIP auf den nächsten Befehl: FMUL. Dieser lädt die Konstante 4.1 aus dem Speicher, wie OllyDbg zeigt.

Dann: FMUL wurde ausgeführt, sodass das Produkt jetzt in ST(0) liegt.

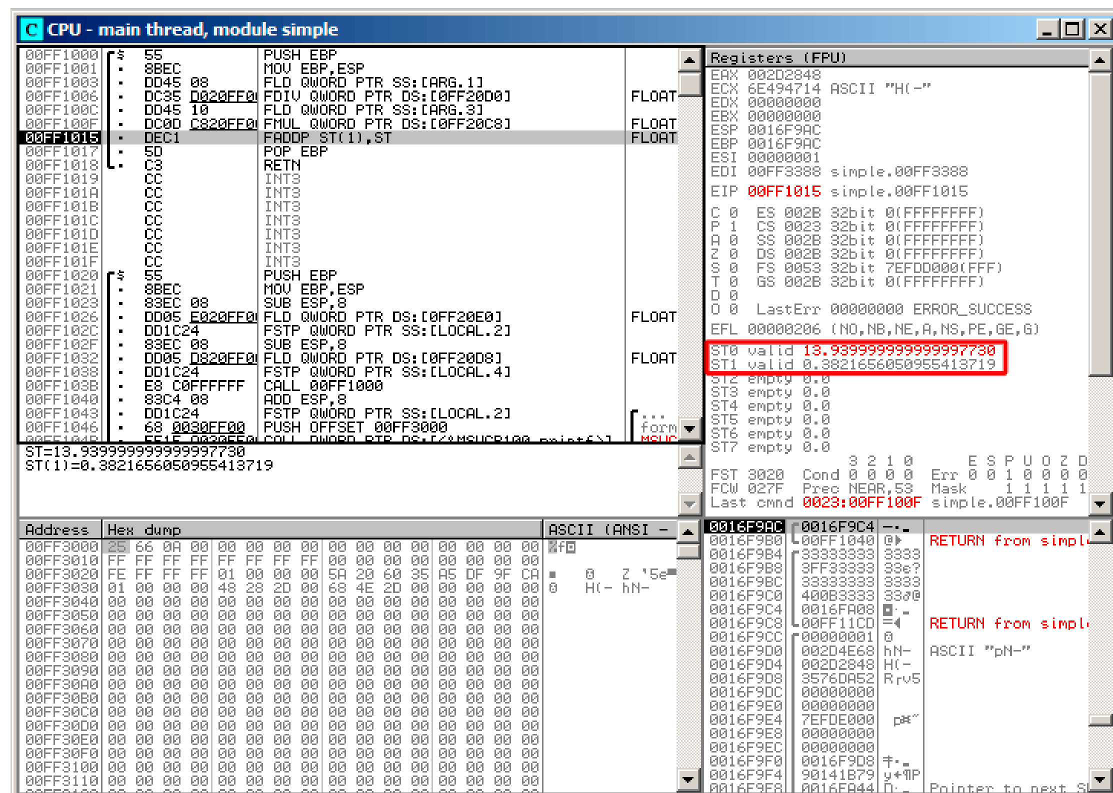


Abbildung 1.65: OllyDbg: FMUL wurde ausgeführt

Dann: der Befehl FADDP wurde ausgeführt, sodass sich in ST(0) nunmehr das Ergebnis der Addition befindet und ST(1) wird gelöscht:

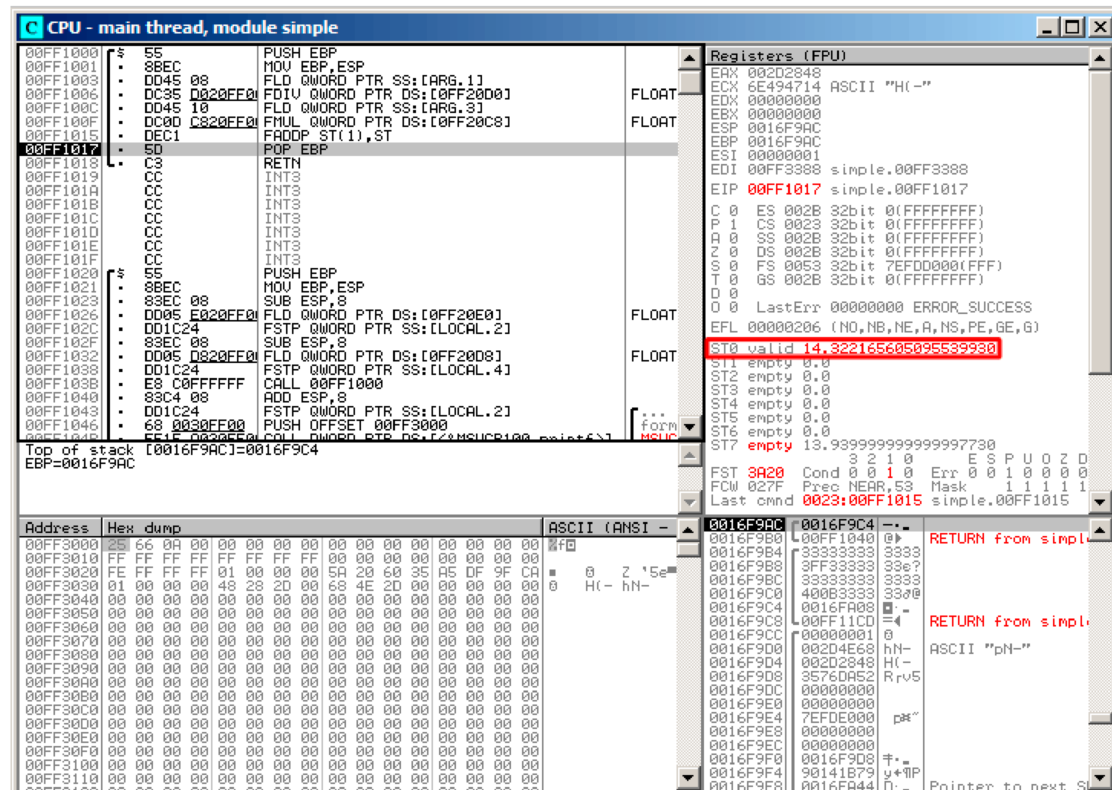


Abbildung 1.66: OllyDbg: FADDP wurde ausgeführt

Das Ergebnis bleibt in ST(0), denn die Funktion liefert ihren Rückgabewert über ST(0) zurück.

Später liest main() diesen Wert aus dem Register.

Wir sehen außerdem etwas Ungewöhnliches: der Wert 13.93...befindet sich nun in ST(7). Warum?

Wie bereits vorher in diesem Buch beschrieben, bilden die FPU einen Stack: 1.19.2 on page 254. Dabei handelt es sich jedoch um eine vereinfachte Darstellung.

Stellen wir uns vor, es wäre genau wie beschrieben in *Hardware* implementiert, dann müssten die Inhalte aller 7 Register während der push und pop-Befehle in jeweils benachbarte Register verschoben (oder kopiert) werden und das würde eine Menge Aufwand bedeuten.

In Wirklichkeit hat die FPU nur 8 Register und einen Pointer (TOP genannt), der die Registernummer enthält, die derzeit oben auf dem Stack liegt.

Wenn ein Wert auf dem Stack abgelegt wird, zeigt TOP auf das nächste verfügbare Register und dann wird der Wert in dieses Register geschrieben.

Dieser Vorgang läuft umgekehrt ab, wenn ein Wert vom Stack geholt wird, aber das freigewordene Register wird nicht gelöscht (es könnnte möglicherweise gelöscht werden, aber dies würde einen Mehraufwand bedeuten und die Performance herabsetzen). Genau das sehen wir hier.

Man kann sagen, dass FADDP die Summe auf dem Stack gespeichert hat und dann ein Element vom Stack geholt hat.

Aber in Wirklichkeit hat der Befehl die Summe gespeichert und dann TOP verschoben.

Genauer gesagt bilden die Register der FPU einen Ringpuffer.

GCC

GCC 4.4.1 (mit der Option -O3) erzeugt fast den gleichen Code, nur leicht verändert.

Listing 1.181: Optimierender GCC 4.4.1

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; Stand des Stacks: ST(0) = 3.14

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; Aktueller Stand des Stacks: ST(0) = Ergebnis der Division

                fld     ds:dbl_8048610 ; 4.1

; Aktueller Stand des Stacks: ST(0) = 4.1, ST(1) = Ergebnis der Division

                fmul    [ebp+arg_8]

; Aktueller Stand des Stacks: ST(0) = Ergebnis der Multiplikation, ST(1) =
; Ergebnis der Division

                pop     ebp
                faddp   st(1), st

; Aktueller Stand des Stacks: ST(0) = Ergebnis der Addition

                retn
f               endp
```

Der Unterschied besteht darin, dass zuerst 3.14 auf dem Stack (in ST(0)) abgelegt wird und danach der Wert in arg_0 durch den Wert in ST(0) geteilt wird.

FDIVR steht für *Reverse Divide* –teilen, wobei Dividend und Divisor miteinander vertauscht werden. Da es sich bei der Multiplikation um eine kommutative Operation handelt, gibt es keinen vergleichbaren Befehl für die Multiplikation. Wir haben es lediglich FMUL ohne -R Gegenstück zur Verfügung.

FADDP addiert die beiden Werte und holt auch einen Wert vom Stack. Nach der Ausführung steht die Summe in ST(0).

ARM: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Bis die Unterstützung für Fließkommaarithmetik in ARM standardisiert wurde, fügten einige Hersteller von Prozessoren ihre eigenen Befehlserweiterungen hinzu. Schließlich wurde VFP (*Vector Floating Point*) standardisiert.

Ein wichtiger Unterschied zum x86 ist, dass in es in ARM keinen Stack gibt, sondern man nur mit den Registern arbeitet.

Listing 1.182: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; lade "a"
                VMOV            D18, R2, R3 ; lade "b"
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF: f
dbl_2CA0        DCFD 4.1                ; DATA XREF: f+10
```

Hier sehen wir, dass einige neue Register mit einem D als Präfix verwendet werden.

Bei diesen handelt es sich um 64-bit-Register; es gibt 32 von ihnen und sie können sowohl für Fließkommazahlen (doppelte Genauigkeit (double)) als auch für SIMD (heißt hier in ARM NEON) benutzt werden.

Es gibt also 32 32-bit-S-Register vorgesehen für Fließkommazahlen in einfacher Genauigkeit (float).

Es ist leicht zu merken: D-Register sind für Zahlen in doppelter Genauigkeit, während S-Register für einfache Genauigkeit (engl. single) vorgesehen sind. Mehr dazu hier:**??** on page ??

Beide Konstanten (3.14 und 4.1) werden im IEEE 754 Format im Speicher abgelegt.

Wie man leicht sieht sind VLDR und VMOV analog zu den LDR und MOV Befehlen, aber arbeiten auf D-Registern.

Es muss angemerkt werden, dass diese Befehle genau wie die D-Register nicht nur für Fließkommazahlen vorgesehen sind, sondern ebenfalls für SIMD (NEON) Operationen verwendet werden können, was wir im folgenden zeigen werden.

Die Paraemter werden der Funktion auf übliche Weise über die R-Register übergeben, aber da jede Zahl in doppelter Genauigkeit eine Größe von 64 Bit hat werden jeweils zwei R-Register benötigt, um eine Zahl zu übergeben.

Der Befehl `VMOV D17, R0, R1` zu Beginn, fasst zwei 32-Bit-Werte aus `R0` und `R1` zu einem 64-Bit-Wert zusammen und speichert diesen in `D17`.

`VMOV R0, R1, D16` ist die umgekehrte Operation: was vorher in `D16` war, wird in zwei Register, `R0` und `R1` aufgeteilt, denn eine Zahl in doppelter Genauigkeit, die 64 Bit Speicherplatz benötigt, wird über `R0` und `R1` zurückgegeben.

`VDIV`, `VMUL` und `VADD` sind Befehle zur Verarbeitung von Fließkommazahlen, die Quotient, Produkt bzw. Summe berechnen.

Der Code für Thumb-2 ist identisch.

ARM: Optimierender Keil 6/2013 (Thumb Modus)

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666 ; 4.1
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F ; 3.14
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

; 4.1 im IEEE 754 Format:
dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
; 3.14 im IEEE 754 Format:
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

Keil erzeugte Code für einen Prozessor ohne FPU oder NEON Unterstützung.

Die Fließkommazahlen in doppelter Genauigkeit werden über die üblichen R-Register übergeben und anstelle von FPU-Befehlen werden Programmbibliotheken (wie z.B. `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`) aufgerufen, welche Multiplikation, Division und Addition auf Fließkommazahlen emulieren.

Diese Vorgehensweise ist natürlich langsamer als der FPU-Koprozessor, aber es ist besser als nichts.

Übrigens waren ähnliche FPU-emulierende Programmbibliotheken auch in der x86-Welt sehr beliebt als Koprozessoren selten und teuer waren und nur auf wertvollen Computern installiert waren.

Die Emulation des FPU-Koprozessors wird *soft float* oder *armel* (in der ARM-Welt) genannt, wohingegen die FPU-Befehle des Koprozessors *hard float* oder *armhf* genannt werden.

ARM64: Optimierender GCC (Linaro) 4.9

Sehr kompakter Code:

Listing 1.183: Optimierender GCC (Linaro) 4.9

```
f:
; D0 = a, D1 = b
        ldr     d2, .LC25       ; 3.14
; D2 = 3.14
        fdiv    d0, d0, d2
; D0 = D0/D2 = a/3.14
        ldr     d2, .LC26       ; 4.1
; D2 = 4.1
        fmadd   d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
        ret

; Konstanten im IEEE 754 Format:
.LC25:
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918      ; 4.1
        .word   1074816614
```

ARM64: Nicht optimierender GCC (Linaro) 4.9

Listing 1.184: Nicht optimierender GCC (Linaro) 4.9

```
f:
        sub     sp, sp, #16
        str     d0, [sp,8]      ; speichere "a" in der Register Save Area
        str     d1, [sp]        ; speichere "b" in der Register Save Area
        ldr     x1, [sp,8]
; X1 = a
        ldr     x0, .LC25
```

```
; X0 = 3.14
        fmov    d0, x1
        fmov    d1, x0
; D0 = a, D1 = 3.14
        fdiv    d0, d0, d1
; D0 = D0/D1 = a/3.14

        fmov    x1, d0
; X1 = a/3.14
        ldr     x2, [sp]
; X2 = b
        ldr     x0, .LC26
; X0 = 4.1
        fmov    d0, x2
; D0 = b
        fmov    d1, x0
; D1 = 4.1
        fmul    d0, d0, d1
; D0 = D0*D1 = b*4.1

        fmov    x0, d0
; X0 = D0 = b*4.1
        fmov    d0, x1
; D0 = a/3.14
        fmov    d1, x0
; D1 = X0 = b*4.1
        fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

        fmov    x0, d0 ; \ redundanter Code
        fmov    d0, x0 ; /
        add     sp, sp, 16
        ret
.LC25:
        .word   1374389535      ; 3.14
        .word   1074339512
.LC26:
        .word   1717986918      ; 4.1
        .word   1074816614
```

Nicht optimierender GCC ist geschwätziger. Hier findet eine Menge unnützes Verschieben von Werten statt, inklusive einigem eindeutig redundantem Code (die letzten beiden FMOV Befehle). Vermutlich ist GCC 4.9 noch nicht besonders gut im Erzeugen von ARM64 Code.

Bemerkenswert ist, dass ARM64 64-Bit-Register besitzt und die D-Register ebenfalls 64 Bit breit sind.

Dadurch steht es dem Compiler frei Werte von Typ *double* in GPRs anstelle auf dem lokalen Stack zu speichern. Dies ist in 32-bit-CPUs nicht möglich.

Wiederum kann man als Übung versuchen diese Funktion manuell zu optimieren ohne neue Befehl wie FMADD einzuführen.

1.19.6 Gleitkommazahlen als Argumente übergeben

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

x86

Schauen wir uns an, was wir in MSVC 2010 erhalten:

Listing 1.185: MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8  ; reserviere Speicher für erste Variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; reserviere Speicher für zweite Variable
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; gib Platz für eine Variable frei.

; im lokalen Stack sind hier immer noch 8 Byte für uns reserviert.
; Ergebnis jetzt in ST(0)

 ; verschiebe Ergebnis von ST(0) auf den lokalen Stack für printf():
    fstp   QWORD PTR [esp]
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD und FSTP verschieben Variablen zwischen Datensegment und dem FPU Stack.`pow()`[106] nimmt beide Werte vom Stack der FPU und gibt ihr Ergebnis über das `ST(0)` Regis-

[106] eine Standard-C-Funktion, die eine Zahl potenziert

ter zurück. Die Funktion `printf()` nimmt 8 Byte vom lokalen Stack und interpretiert diese als Variable von Typ *double*.

Übrigens könnte hier auch ein Paar `MOV` Befehle verwendet werden, um die Werte aus dem Speicher zu holen und auf den Stack zu legen, denn die Werte sind im Speicher im IEEE 754 Format abgelegt und pow() arbeitet mit diesem Format, sodass keine Umwandlung notwendig ist. Genau so wird es im folgenden Beispiel für ARM auch gemacht:1.19.6

ARM + Nicht optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
_main

var_C     = -0xC

          PUSH     {R7,LR}
          MOV      R7, SP
          SUB      SP, SP, #4
          VLDR     D16, =32.01
          VMOV     R0, R1, D16
          VLDR     D16, =1.54
          VMOV     R2, R3, D16
          BLX      _pow
          VMOV     D16, R0, R1
          MOV      R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
          ADD      R0, PC
          VMOV     R1, R2, D16
          BLX      _printf
          MOVS     R1, 0
          STR      R0, [SP,#0xC+var_C]
          MOV      R0, R1
          ADD      SP, SP, #4
          POP      {R7,PC}

dbl_2F90  DCFD 32.01       ; DATA XREF: _main+6
dbl_2F98  DCFD 1.54        ; DATA XREF: _main+E
```

Wie bereits vorher erwähnt werden Pointer auf 64-Bit-Fließkommazahlen über ein Paar von R-Registern übergeben.

Dieser Code ist leicht redundant (sicherlich aufgrund der deaktivierten Optimierung), da es möglich ist Werte direkt in die R-Register zu laden, ohne die D-Register zu verwenden.

Wie wir also sehen erhält die _pow Funktion ihr erster Argument in `R0` und R1 und das zweite in R2 und R3. Die Funktion speichert ihr Ergebnis in `R0` und R1. Das Ergebnis von _pow wird zunächst nach D16 und anschließend in das Paar R1 und R2 verschoben, von wo aus `printf()` das Ergebnis übernimmt.

ARM + Nicht optimierender Keil 6/2013 (ARM Modus)

```
_main
```

```
      STMFD   SP!, {R4-R6,LR}
      LDR     R2, =0xA3D70A4  ; y
      LDR     R3, =0x3FF8A3D7
      LDR     R0, =0xAE147AE1 ; x
      LDR     R1, =0x40400147
      BL      pow
      MOV     R4, R0
      MOV     R2, R4
      MOV     R3, R1
      ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
      BL      __2printf
      MOV     R0, #0
      LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF: _main+8
x               DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528       DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF: _main+24
```

Die D-Register werden hier nicht verwendet, sondern nur Paare von R-Registern.

ARM64 + Optimierender GCC (Linaro) 4.9

Listing 1.186: Optimierender GCC (Linaro) 4.9

```
f:
        stp     x29, x30, [sp, -16]!
        add     x29, sp, 0
        ldr     d1, .LC1 ; lade 1.54 nach D1
        ldr     d0, .LC0 ; lade 32.01 nach D0
        bl      pow
; Ergebnis von pow() in D0
        adrp    x0, .LC2
        add     x0, x0, :lo12:.LC2
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
.LC0:
; 32.01 im IEEE 754 Format
        .word   -1374389535
        .word   1077936455
.LC1:
; 1.54 im IEEE 754 Format
        .word   171798692
        .word   1073259479
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
```

Die Konstanten werden nach D0 und D1 geladen: pow() übernimmt sie von dort. Das Ergebnis befindet sich nach der Ausführung von pow() in D0. Es wird ohne weite-

re Änderung oder Verschiebung an die Funktion `printf()` übergeben, da `printf()` ganzzahlige Werte und Pointer aus X-Registern, Fließkommaparameter jedoch aus D-Registern übernimmt.

1.19.7 Vergleichsoperation

Versuchen wir folgendes:

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

Obwohl die Funktion einfach ist, wird es schwierig werden zu verstehen wie sie funktioniert.

x86

Nicht optimierender MSVC

MSVC 2010 erzeugt den folgenden Code:

Listing 1.187: Nicht optimierender MSVC 2010

```
PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8                ; size = 8
_b$ = 16               ; size = 8
_d_max     PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _b$[ebp]

; Zustand des Stacks: ST(0) = _b
; vergleiche _b (ST(0)) und _a, und hole Register vom Stack

    fcomp  QWORD PTR _a$[ebp]

; Stack ist jetzt leer

    fnstsw ax
```

```
    test   ah, 5
    jp     SHORT $LN1@d_max

; hierher gelangen wir nur, falls a>b

    fld    QWORD PTR _a$[ebp]
    jmp    SHORT $LN2@d_max
$LN1@d_max:
    fld    QWORD PTR _b$[ebp]
$LN2@d_max:
    pop    ebp
    ret    0
_d_max     ENDP
```

Der Befehl FLD lädt _b nach ST(0).

FCOMP verlgeicht den Wert in ST(0) mit dem Wert, der sich in _a befindet und setzt die C3/C2/C0 im FPU Status Register entsprechend. Das Statusregister ist ein 16-Bit-Register, das den aktueller Zustand der FPU abbildet.

Nachdem die Bits gesetzt worden sind, nimmer der FCOMP Befehl auch eine Variable vom Stack. Dieses Verhalten unterscheidet ihn von FCOM, der einfach zwei Werte vergleicht und den Stack unangetastet lässt.

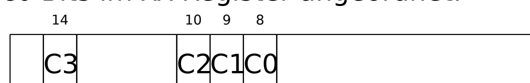
Leider verfügen CPUs vor Intel P6[107]über keinerlei bedingte Sprungbefehle, die die C3/C2/C0 prüfen.

After the bits are set, the FCOMP instruction also pops one variable from the stack. This is what distinguishes it from FCOM, which is just compares values, leaving the stack in the same state. Vielleicht ist diese Tatsache historisch begründet (man erinnere sich: die FPU war früher ein eigener Chip).
Moderne CPUs, beginnend mit Intel P6 haben FCOMI/FCOMIP/FUCOMI/FUCOMIP Befehle –welche im Prinzip das gleiche tun, aber die ZF/PF/CF Flags der CPU verändern können.

Der FNSTSW Befehl kopiert das FPU Statusregister nach AX. C3/C2/C0 werden an den Stellen 14/10/8 abgelegt, sie befinden sich im AX Register an den gleichen Stellen und sie werden alle in höherwertigen Teil von AX —AH abgelegt.

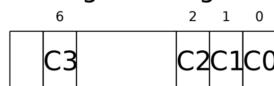
- Falls in unserem Beispiel $b > a$, dann werden die C3/C2/C0 Bits wie folgt gesetzt: 0, 0, 0.
- Falls $a > b$, dann ist das Bitmuster: 0, 0, 1.
- Falls $a = b$, dann ist das Bitmuster: 1, 0, 0.
- Wenn das Ergebnis (z.B. im Fehlerfall) ungeordnet ist, dann werden die Bits wie folgt gesetzt: 1,1,1.

So werden die C3/C2/C0 Bits im AX Register angeordnet:



[107]Intel P6 ist Pentium Pro, Pentium II, etc.

So werden die C3/C2/C0 Bits im AH Register angeordnet:



Nach der Ausführung von `test ah, 5`[108] werden nur die C0 und C2 Bits (an den Stellen 0 und 2) betrachtet, alle übrigen Bits werden einfach überlesen.

Werfen wir nun einen Blick auf ein anderes bemerkenswertes historisches Überbleibsel: das *parity flag*.

Dieses Flag wird auf 1 gesetzt, falls die Anzahl der Einsen im Ergebnis der letzten Berechnung gerade ist und auf 1, falls dies nicht der Fall ist.

Schlagen wir in der Wikipedia nach[109]:

> Ein guter Grund das Parity Flag abzufragen, hat tatsächlich gar nichts mit Parität zu tun. Die FPU hat vier Bedingungsflags (C0 bis C3), aber diese können nicht direkt abgefragt werden, sondern müssen zunächst in das Flags Register kopiert werden. Wenn dies geschieht, wird C0 im Carry Flag abgelegt, C2 im Parity Flag und C3 im Zero Flag. Das C2 Flag ist gesetzt, wenn z.B. unvergleichbare Fließkommawerte (NaN oder nicht unterstütztes Format) über der FUCOM Befehl miteinander verglichen werden.*(Übersetzung aus der englischen Wikipedia.)*

Wie in der Wikipedia dargestellt wird das Parity Flag manchmal im FPU Code verwendet; schauen wir uns genauer an wie das funktioniert.

Das PF Flag wird auf 1 gesetzt, wenn sowohl C0 als auch C2 beide 0 oder beide 1 sind. In diesem Fall wird der nachfolgende Sprung JP (*jump if PF==1*) ausgeführt. Wenn wir die Werte der C3/C2/C0 in den unterschiedlichen Fällen betrachten, dann sehen wir, dass der bedingte Sprung JP in zwei Fällen ausgeführt wird: wenn $b > a$ oder wenn $a = b$ (das C3 Bit wird hier nicht betrachtet, da es durch den Befehl `test ah,5`) gelöscht wurde).

Der Rest ist leicht nachvollziehbar. Denn der bedingte Sprung ausgeführt wurde, lädt FLD den Wert von _b nach ST(0) und wenn nicht, wird der Wert von _a dorthin geladen.

Was ist mit der Abfrage von C2?

Das C2 Flag wird im Fehlerfall (NaN, etc.) gesetzt, aber unser Code prüft dies nicht. Wenn sich der Programmierer für FPU Fehler interessiert, muss er zusätzliche Abfragen hinzufügen.

[108] 5=101b
[109] https://en.wikipedia.org/wiki/Parity_flag

Erstes OllyDbg Beispiel: a=1.2 und b=3.4

Laden wir das Beispiel in OllyDbg:

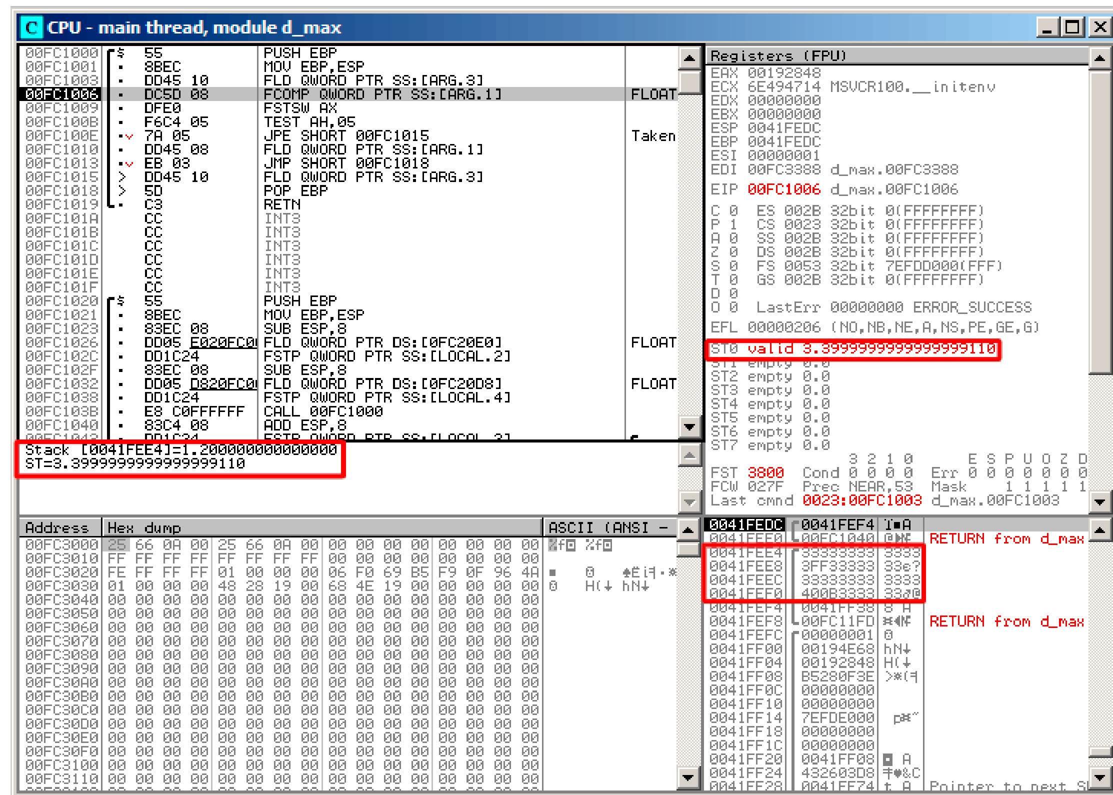


Abbildung 1.67: OllyDbg: erstes FLD wurde ausgeführt

Die aktuellen Parameter der Funktion sind: $a = 1.2$ und $b = 3.4$ (Wir finden sie auf dem Stack zwei 32-Bit-Werte). b (3.4) wurde bereits nach `ST(0)` geladen. Jetzt wird `FCOMP` ausgeführt. OllyDbg zeigt das zweite Argument von `FCOMP`, welches sich jetzt auf dem Stack befindet.

FCOMP wurde ausgeführt:

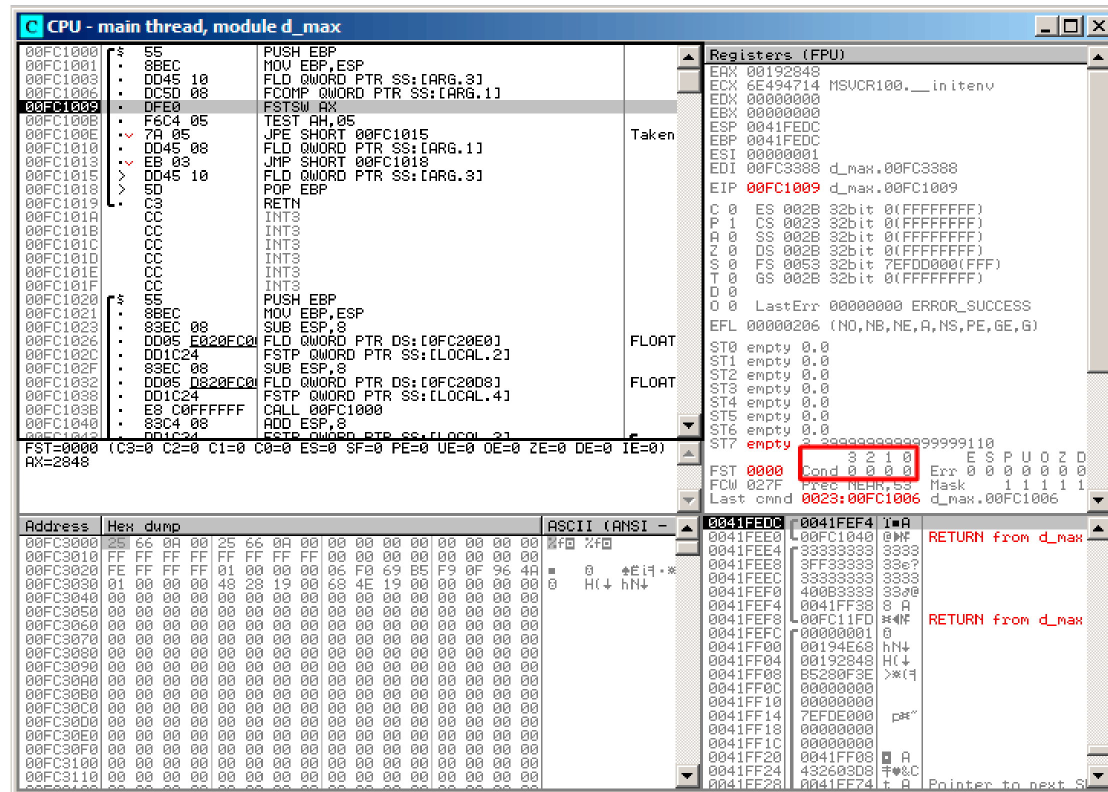


Abbildung 1.68: OllyDbg: FCOMP wurde ausgeführt

Wir sehen den Status der Flags der FPU: sämtlich null. Den geholten Wert finden wir unter ST(7) wieder, auf die Gründe dafür sind wir bereits eingegangen:1.19.5 on page 261.

FNSTSW wurde ausgeführt:

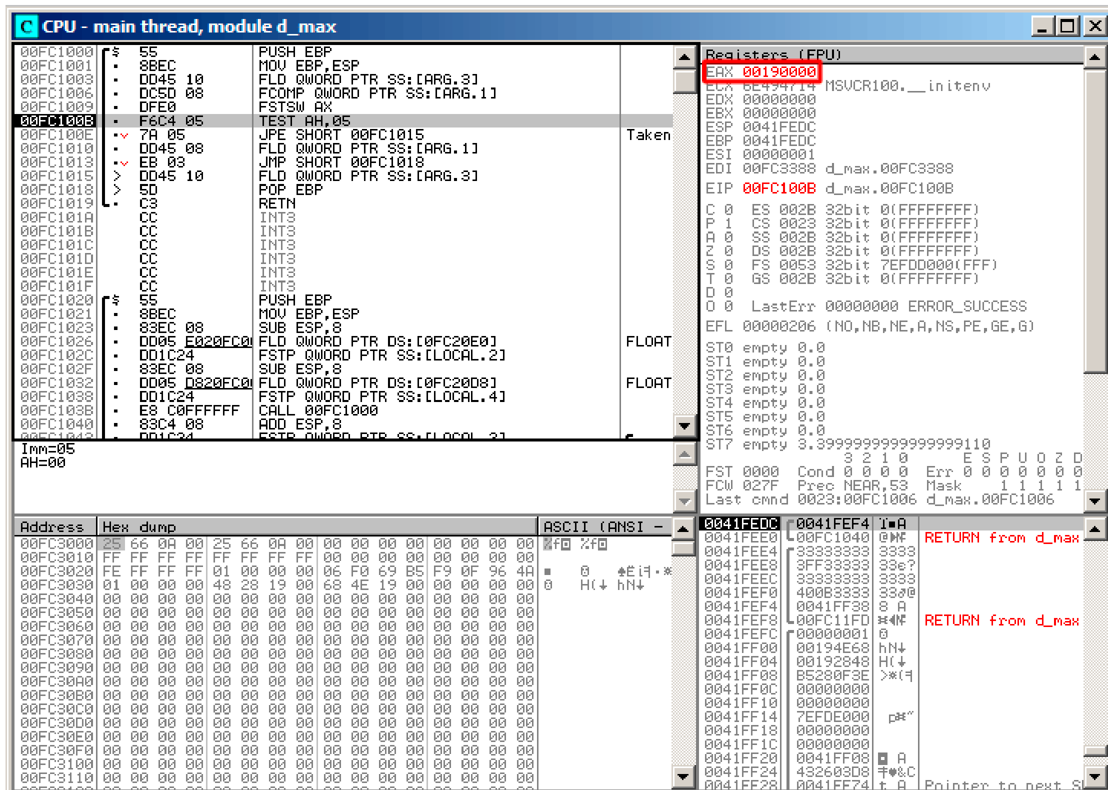


Abbildung 1.69: OllyDbg: FNSTSW wurde ausgeführt

Wir sehen, dass das AX Register Nullen enthält: das passt, da alle Flag auf Null gesetzt sind. (OllyDbg disassembliert hier den Befehl FNSTSW als FSTSW–die beiden sind synonym).

TEST wurde ausgeführt:

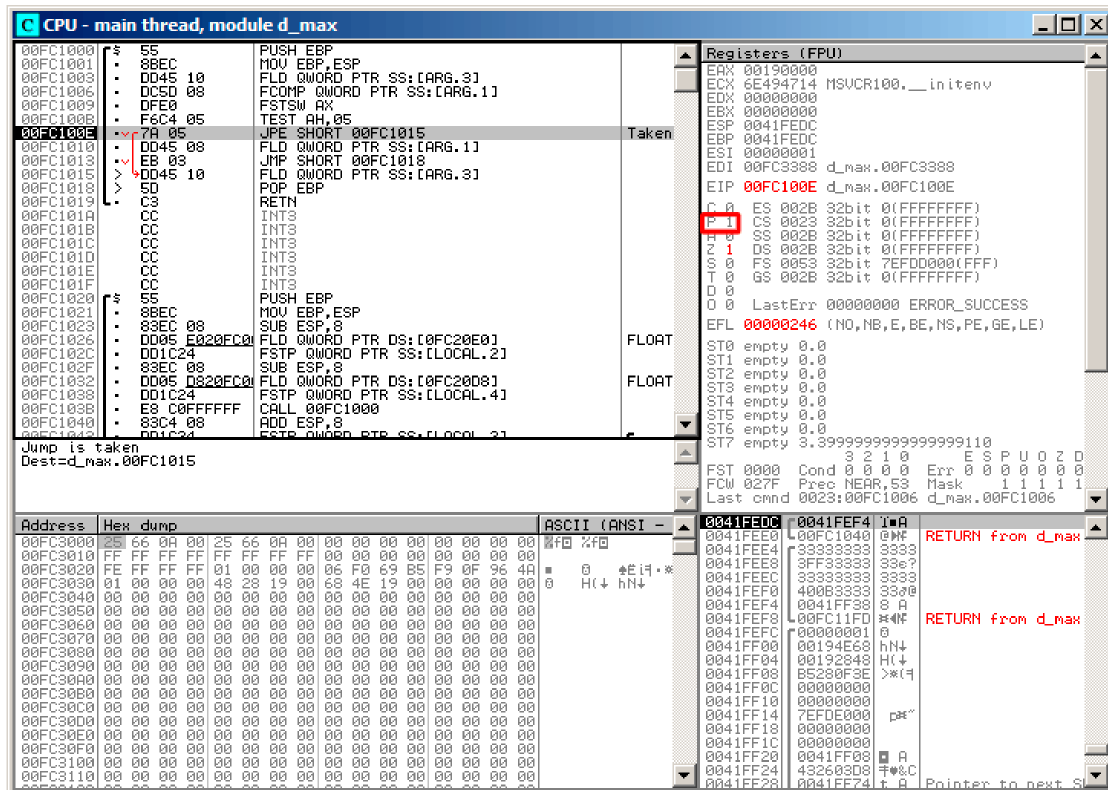


Abbildung 1.70: OllyDbg: TEST wurde ausgeführt

Das PF Flag ist auf 1 gesetzt.

Begründung: die Anzahö der auf null gesetzten Bits ist null, und Null ist eine gerade Zahl. OllyDbg disassembliert JP als JPE[110]–die beiden sind synonym–und dieser Befehl wird als nächstes ausgeführt.

[110] Jump Parity Even (x86 Instruktion)

JPE wird ausgeführt, FLD lädt den Wert von b (3.4) nach ST(0):

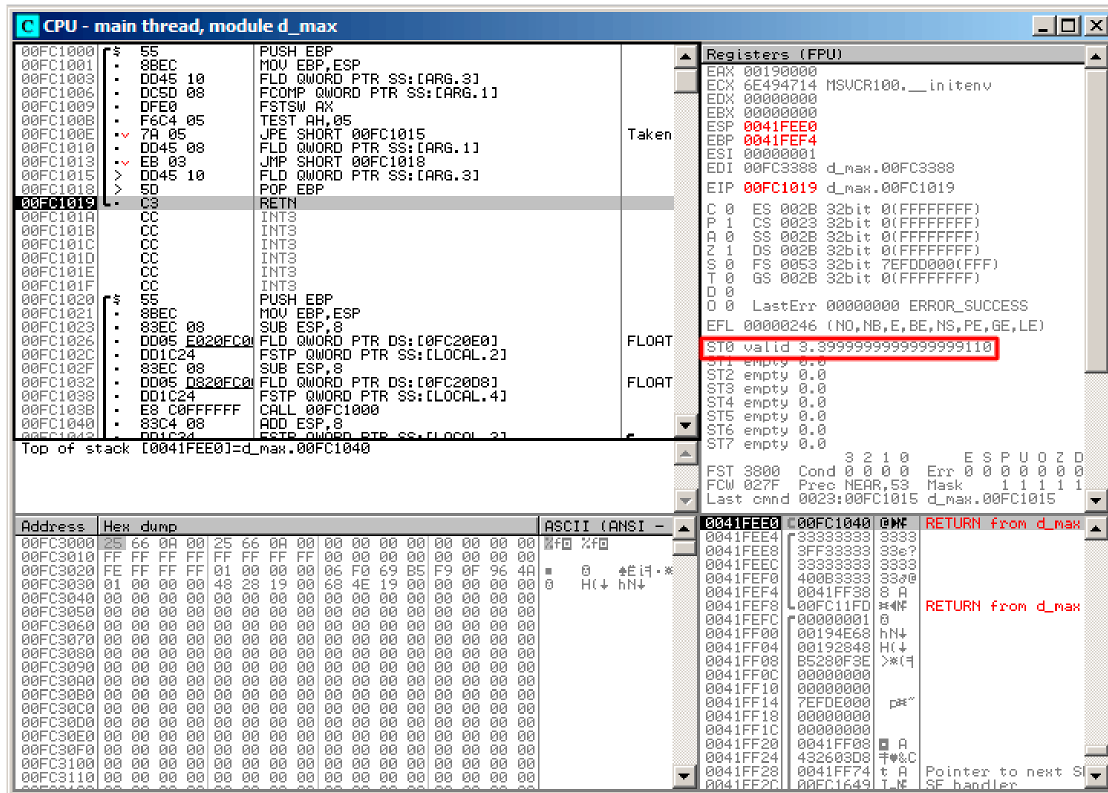


Abbildung 1.71: OllyDbg: das zweite FLD wurde ausgeführt

Die Funktion beendet ihre Arbeit.

Zweites OllyDbg Beispiel: a=5.6 und b=-4

Laden wir unser Beispiel in OllyDbg:

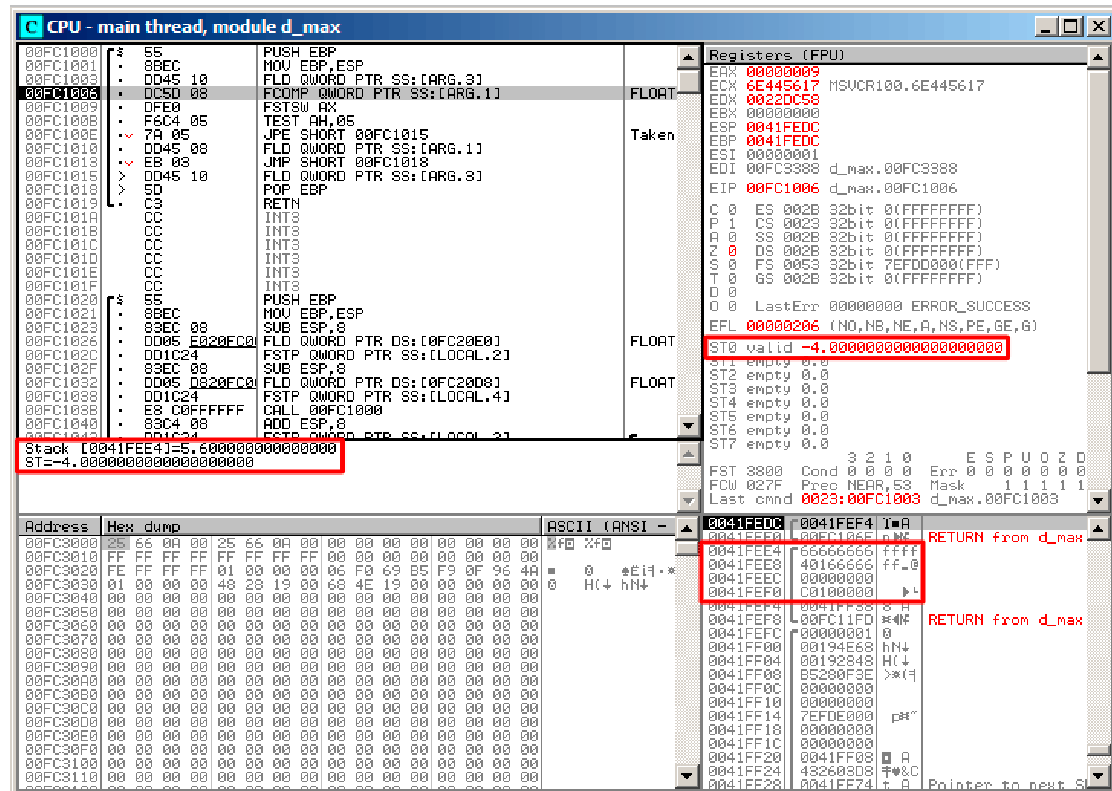


Abbildung 1.72: OllyDbg: erstes FLD ausgeführt

Die aktuellen Funktionsparameter sind: $a = 5.6$ und $b = -4$. b (-4) wurde bereits nach ST(0) geladen. FCOMP wird jetzt ausgeführt. OllyDbg zeigt das zweite Argument von FCOMP, welches sich jetzt auf dem Stack befindet.

FCOMP wird ausgeführt:

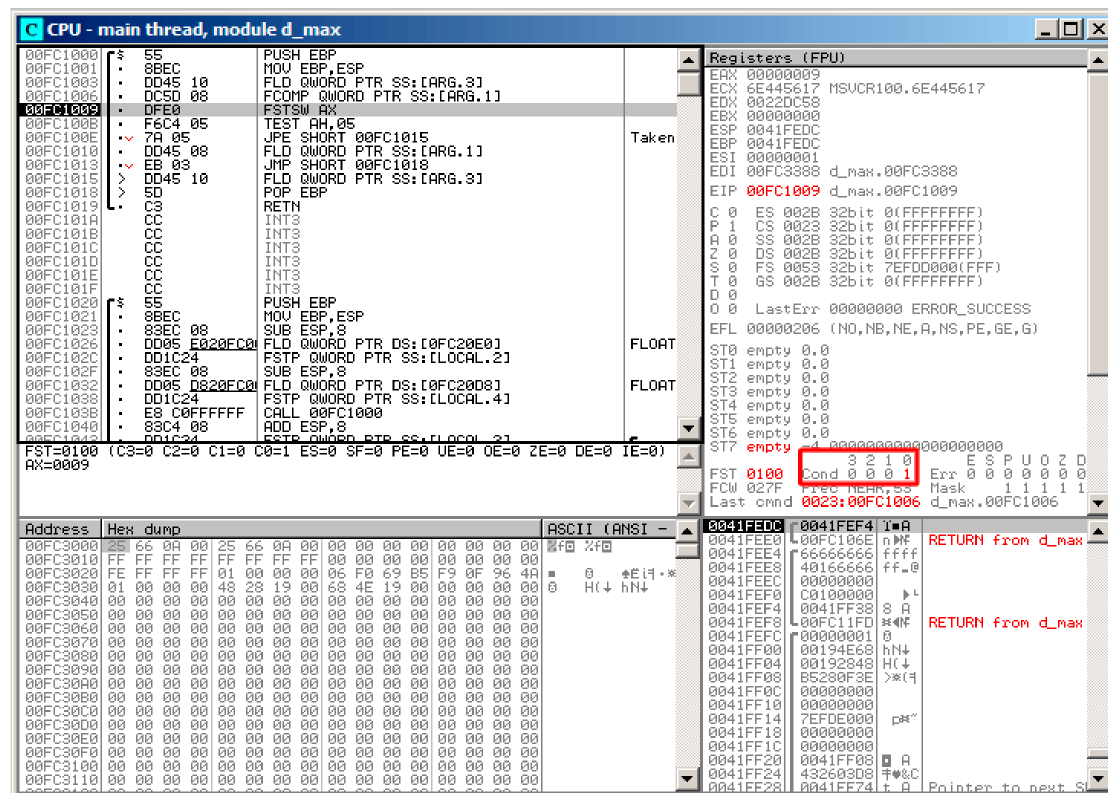


Abbildung 1.73: OllyDbg: FCOMP wird ausgeführt

Wir betrachten den Status der FPU Flas: alle null, außer C0.

FNSTSW wird ausgeführt:

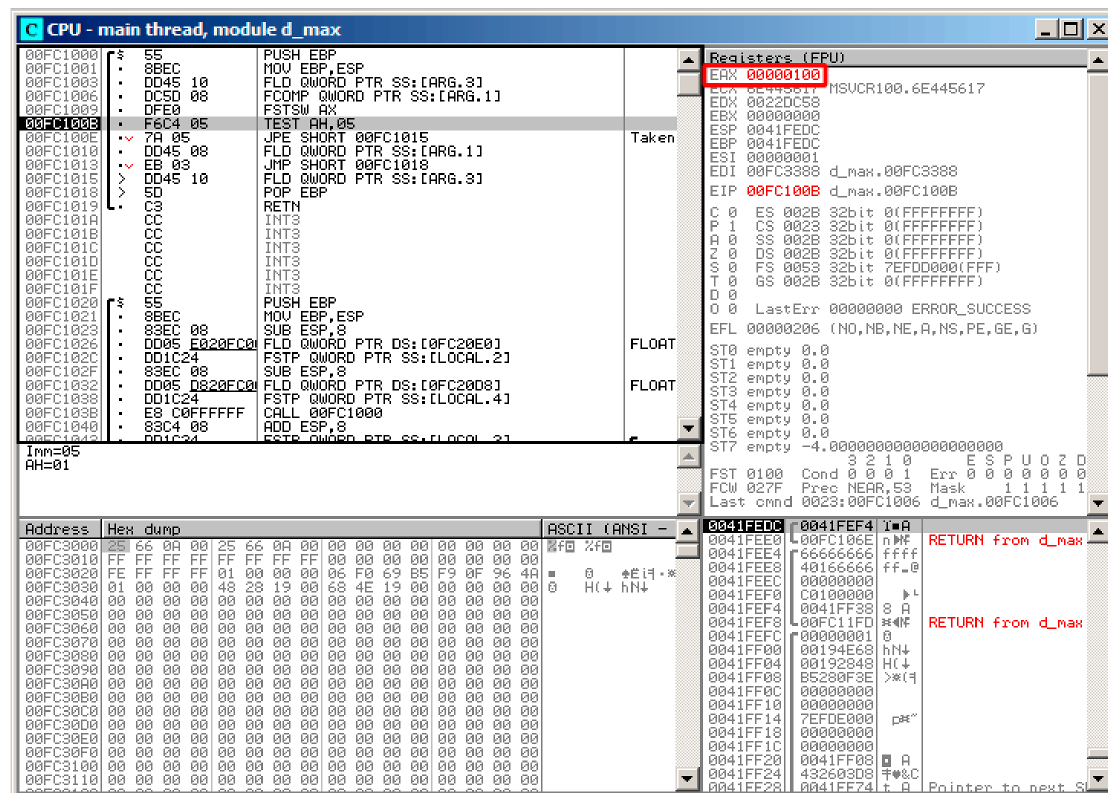


Abbildung 1.74: OllyDbg: FNSTSW wird ausgeführt

Wir sehen, dass das AX Register den Wert `0x100` enthält: das `C0` Flag sitzt auf dem achten Bit.

TEST wird ausgeführt:

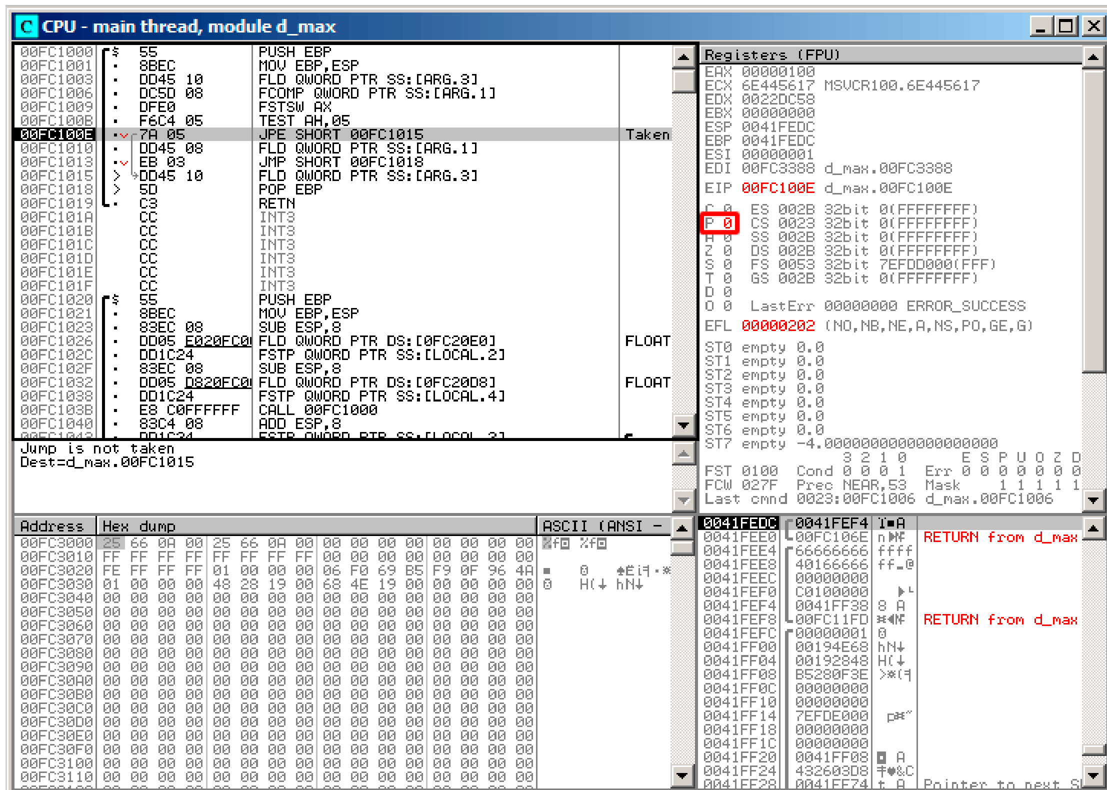


Abbildung 1.75: OllyDbg: TEST wird ausgeführt

Das PF Flag wird gelöscht. Begründung: die Anzahl der gesetzten Bits in `0x100` ist 1 und 1 ist eine ungerade Zahl. JPE wird jetzt übersprungen.

JPE wurde nicht ausgelöst, sodass FLD den Wert von a (5.6) nach ST(0) lädt:

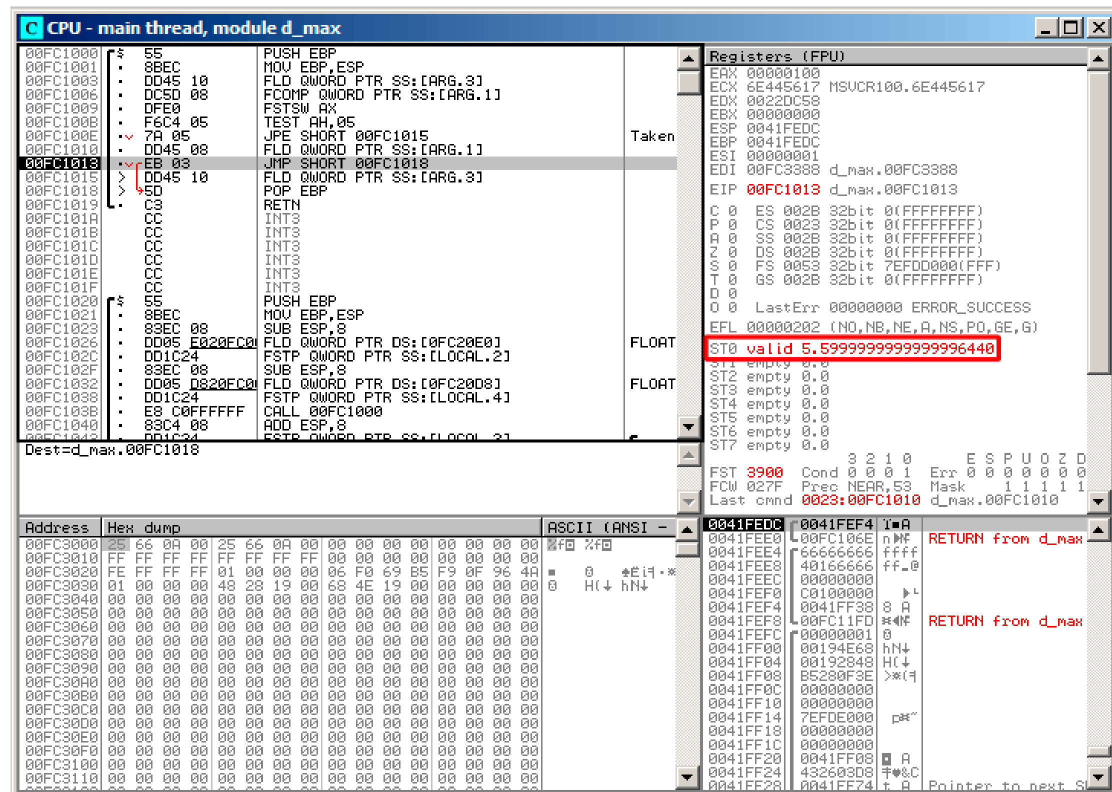


Abbildung 1.76: OllyDbg: zweites FLD wurde ausgeführt

Die Funktion beendet ihre Arbeit.

Optimierender MSVC 2010

Listing 1.188: Optimierender MSVC 2010

```
_a$ = 8                    ; size = 8
_b$ = 16                   ; size = 8
_d_max    PROC
    fld    QWORD PTR _b$[esp-4]
    fld    QWORD PTR _a$[esp-4]

; Zustand des Stacks: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; vergleiche _a und ST(1) = (_b)
    fnstsw  ax
    test    ah, 65 ; 00000041H
    jne     SHORT $LN5@d_max
```

```
; kopiere ST(0) nach ST(1) und hole Register vom Stack,
; lasse (_a) oben auf dem Stack
    fstp    ST(1)

; Zustand des Stacks: ST(0) = _a

    ret    0
$LN5@d_max:
; kopiere ST(0) nach ST(0) und hole Register vom Stack,
; lasse (_b) oben auf dem Stack
    fstp    ST(0)

; Zustand des Stacks: ST(0) = _b

    ret    0
_d_max    ENDP
```

FCOM unterscheidet sich von FCOMP in dem Sinne, dass es nur die Werte vergleicht ohne den FPU Stack zu verändern. Anders als im vorangehenden Beispiel liegen die Operanden hier in umgekehrter Reihenfolge um vor. Dies ist der Grund warum das Ergebnis dieses Vergleichs bezüglich der C3/C2/C0 unterschiedlich ist:

- Falls $a > b$ in unserem Beispiel, dann werden die C3/C2/C0 Bits wie folgt gesetzt: 0, 0, 0.
- Falls $b > a$, dann ist das Bitmuster: 0, 0, 1.
- Falls $a = b$, dann ist das Bitmuster: 1, 0, 0.

Der Befehl `test ah, 65` setzt zwei Bits —C3 und C0. Beide werden auf 0 gesetzt, falls $a > b$: in diesem Fall wird der JNE Sprungbefehl nicht ausgeführt. Dann folgt FSTP ST(1) —dieser Befehl kopiert den Wert von ST(0) in den Operanden und holt einen Wert vom FPU Stack. Mit anderen Worten, der Befehl kopiert ST(0) (in dem sich gerade der Wert von _a befindet) nach ST(1). Anschließend befinden sich zwei Kopien von _a oben auf dem Stack. Nun wird ein Wert wieder vom Stack geholt. Schließlich enthält ST(0) den Wert _a und die Funktion beendet sich.

Der bedingte Sprung JNE wird in zwei Fällen ausgeführt: wenn $b > a$ oder wenn $a = b$. ST(0) wird nach ST(0) kopiert; dabei handelt es sich um eine Operation ohne Wirkung (NOP), dann wird ein Wert vom Stack geholt und in ST(0) steht dann was sich vorher in ST(1) befunden hat, nämlich _b. Danach beendet sich die Funktion. Der Grund dafür, dass dieser Befehl hier erzeugt wird ist wahrscheinlich, dass die FPU über keinen anderen Befehl verfügt um einen Wert vom Stack zu holen und anschließend zu entsorgen.

Erstes OllyDbg Beispiel: a=1.2 und b=3.4

Beide FLD werden ausgeführt:

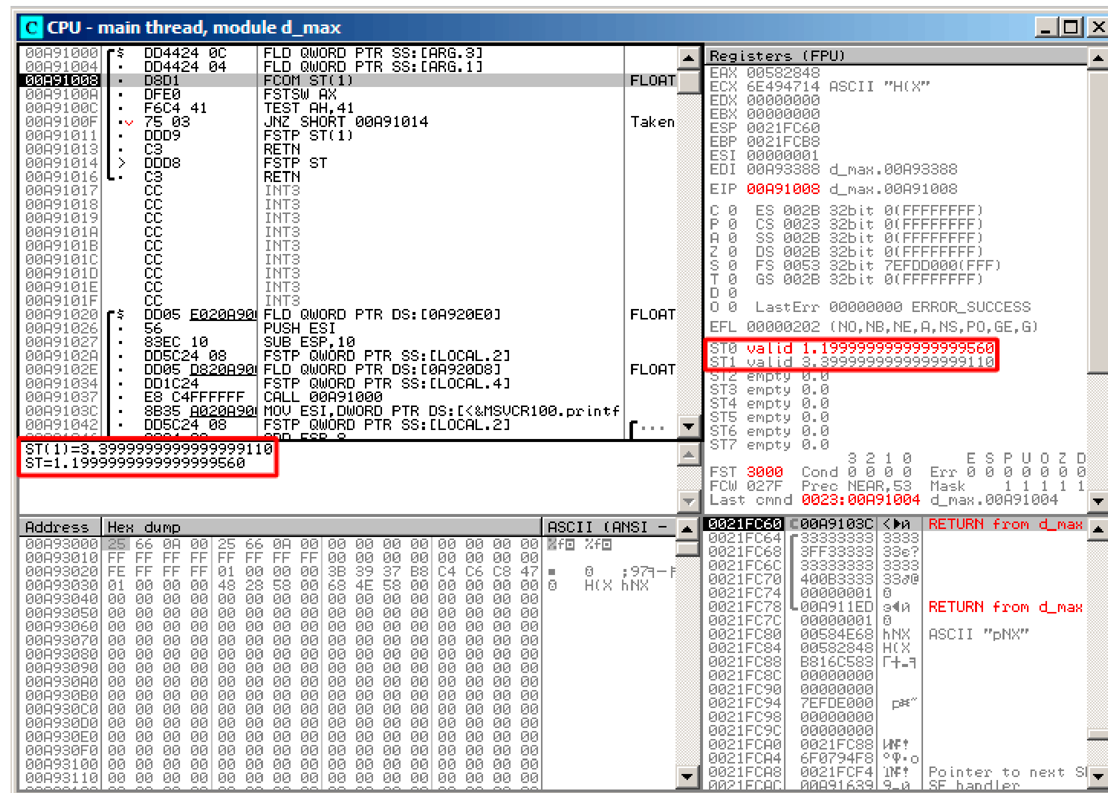


Abbildung 1.77: OllyDbg: beide FLD werden ausgeführt

FCOM wird ausgeführt: OllyDbg zeigt die Inhalte ST(0) und ST(1) übersichtlich an.

FCOM wurde ausgeführt:

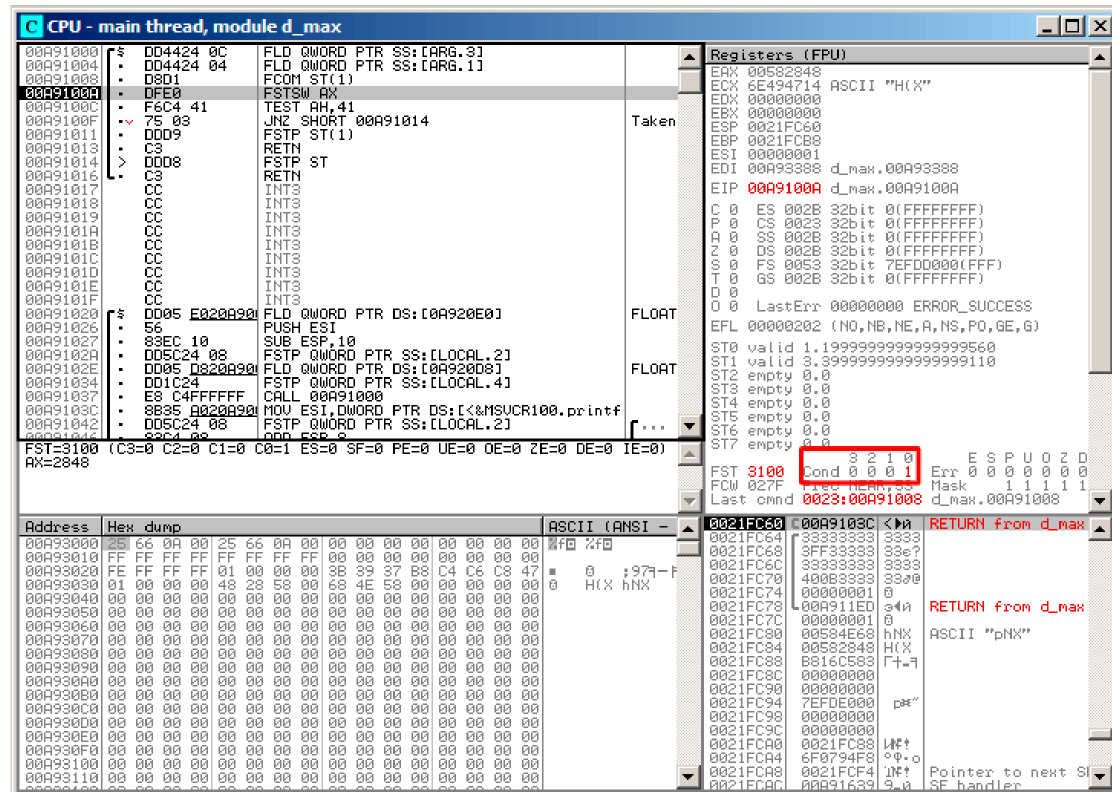


Abbildung 1.78: OllyDbg: FCOM wurde ausgeführt

C0 ist gesetzt, alle anderen Flags sind gelöscht.

FNSTSW wurde ausgeführt, AX=0x3100:

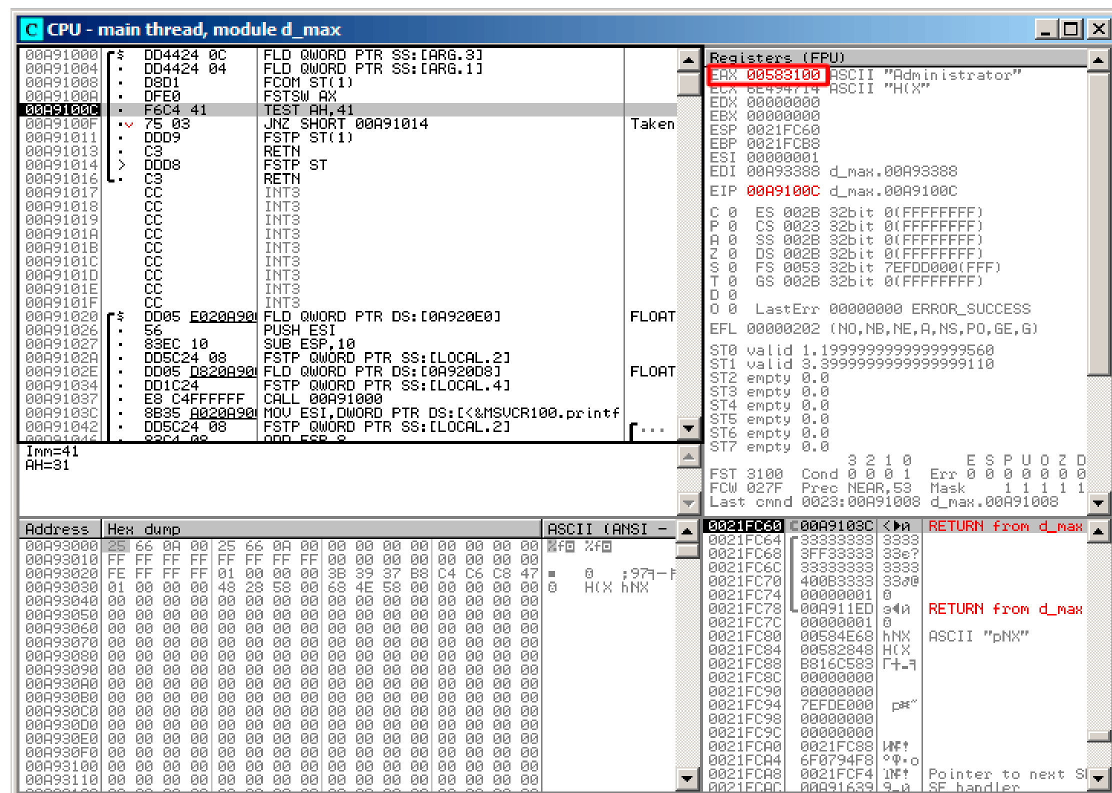


Abbildung 1.79: OllyDbg: FNSTSW wird ausgeführt

TEST wird ausgeführt:

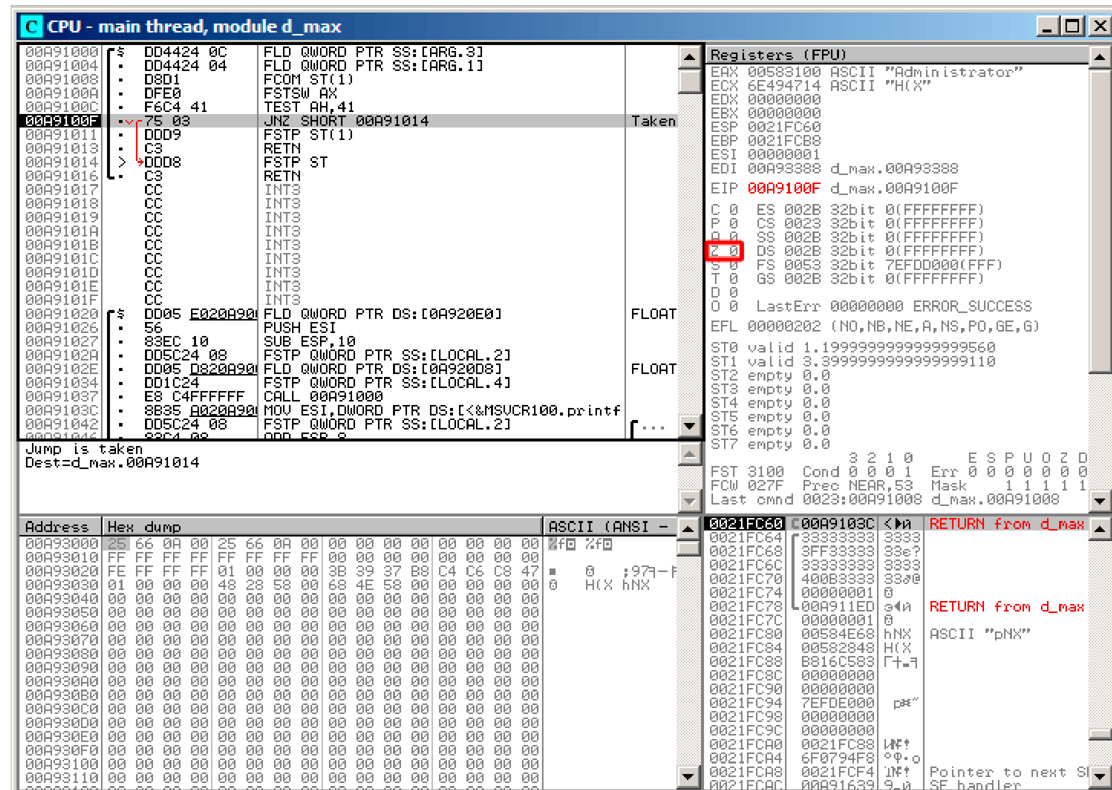


Abbildung 1.80: OllyDbg: TEST wird ausgeführt

ZF=0, conditional bedingter Sprung wird jetzt ausgeführt.

FSTP ST (oder FSTP ST(0)) wurde ausgeführt —1.2 wurde vom Stack geholt und 3.4 bleibt obenauf liegen:

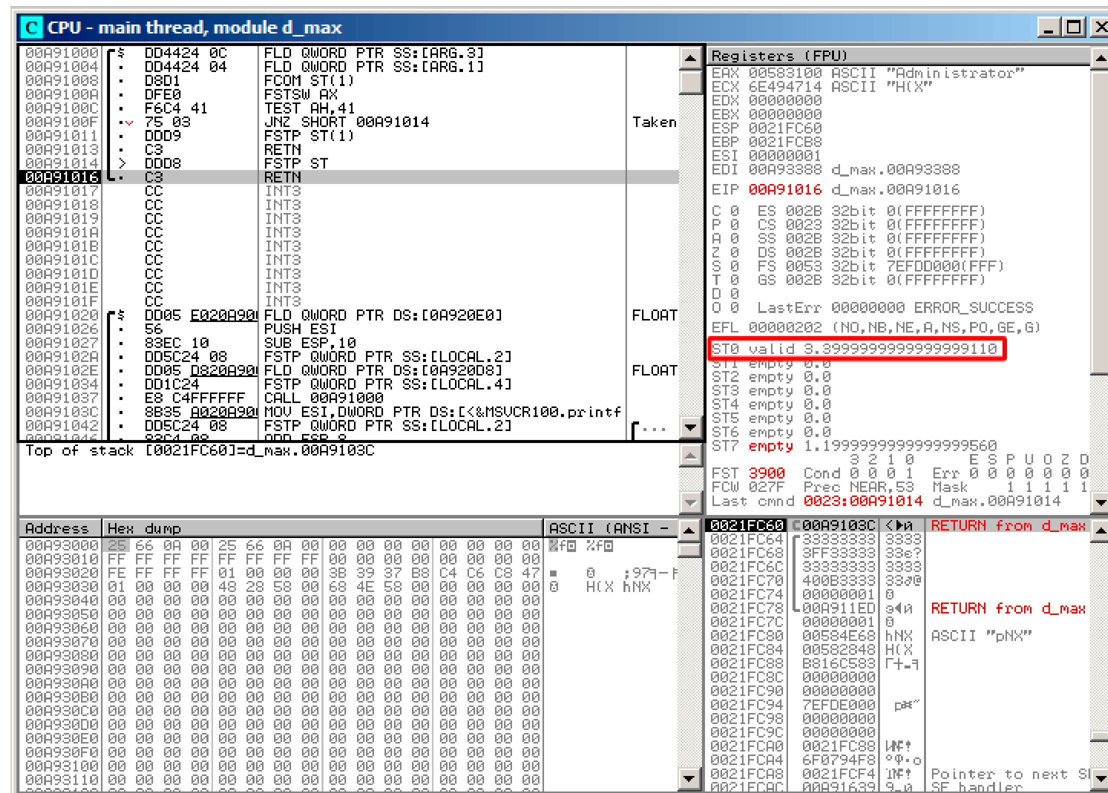


Abbildung 1.81: OllyDbg: FSTP wird ausgeführt

Wir sehen, dass das Ergebnis des Befehls FSTP ST dem Holen eines Wertes vom FPU Stack entspricht.

Second OllyDbg example: a=5.6 and b=-4

Beide FLD werden ausgeführt:

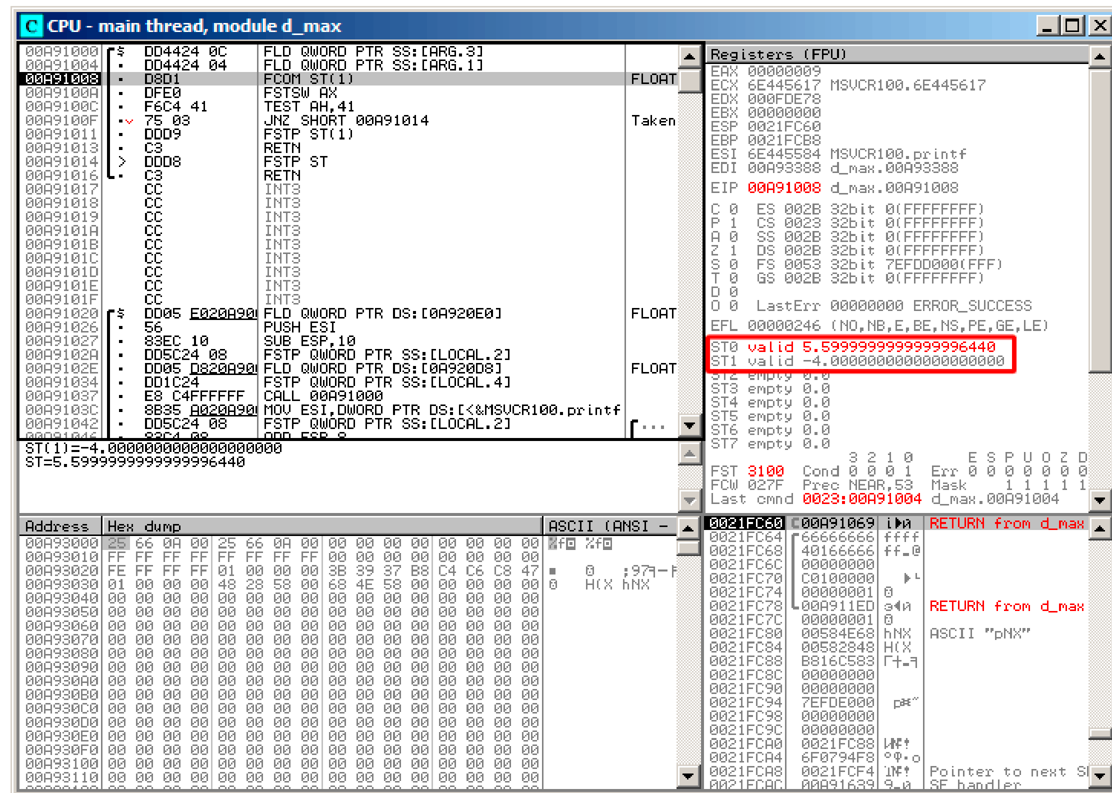


Abbildung 1.82: OllyDbg: beide FLD werden ausgeführt

FCOM wird gleich ausgeführt.

FCOM wurde ausgeführt:

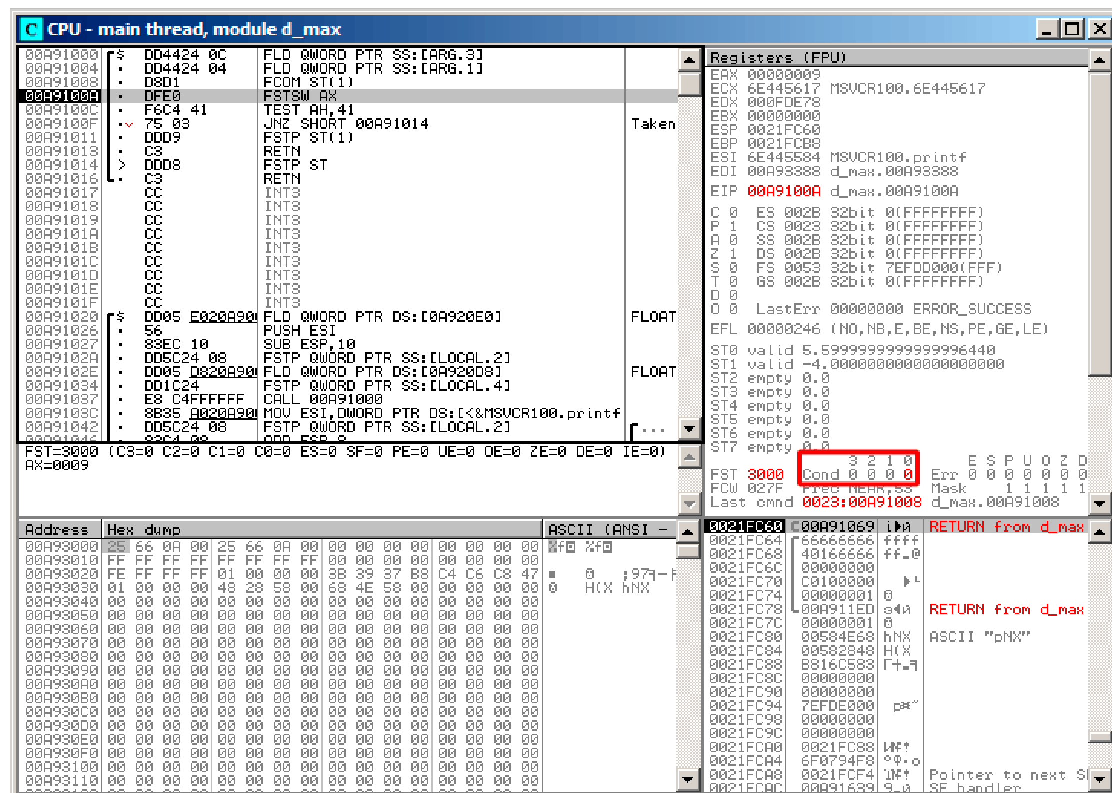


Abbildung 1.83: OllyDbg: FCOM ist beendet

Alle Bedingungs-Flags sind gelöscht.

FNSTSW ist abgearbeitet, AX=0x3000:

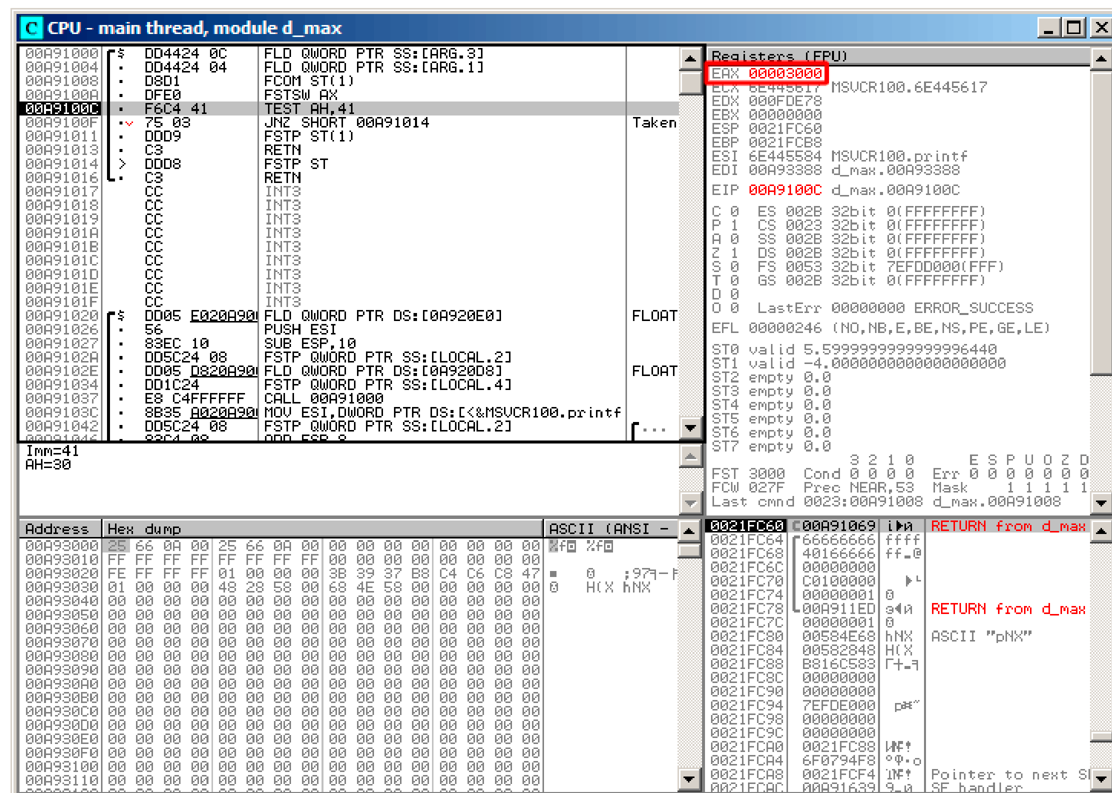


Abbildung 1.84: OllyDbg: FNSTSW wurde ausgeführt

TEST wurde ausgeführt:

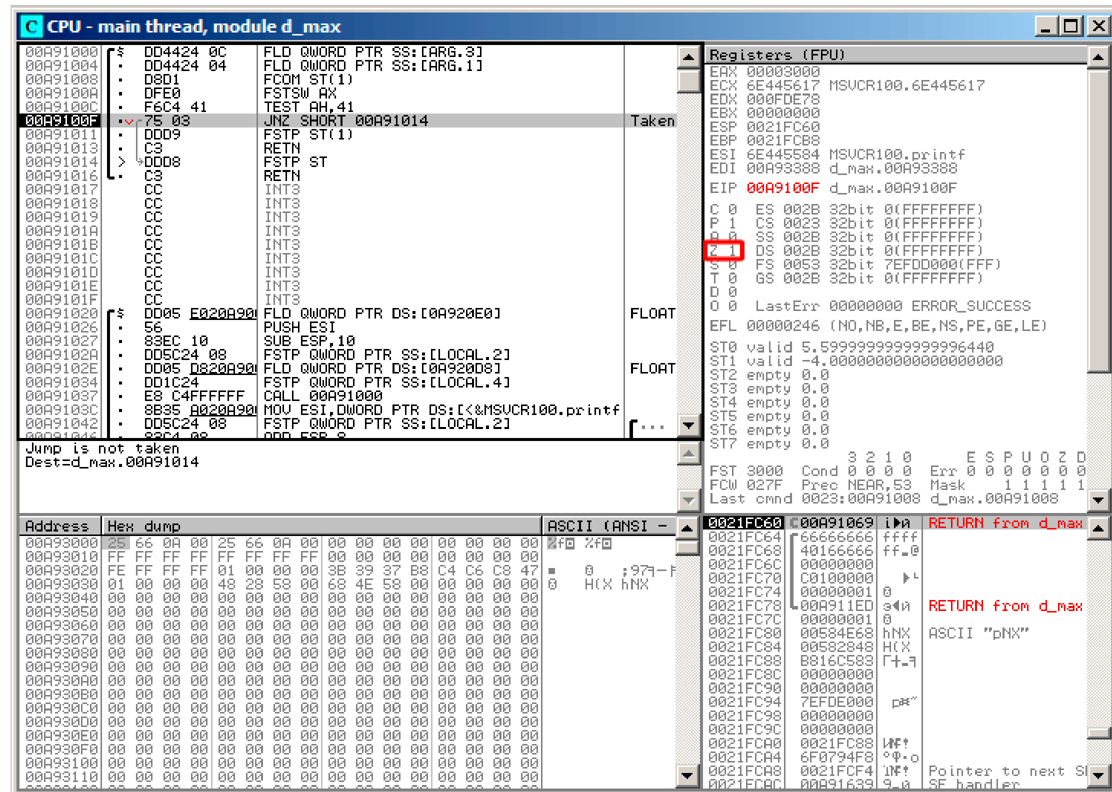


Abbildung 1.85: OllyDbg: TEST wurde ausgeführt

ZF=1, der Sprung wird jetzt nicht ausgeführt.

FSTP ST(1) wurde ausgeführt: der Wert 5.6 liegt jetzt oben auf dem FPU Stack.

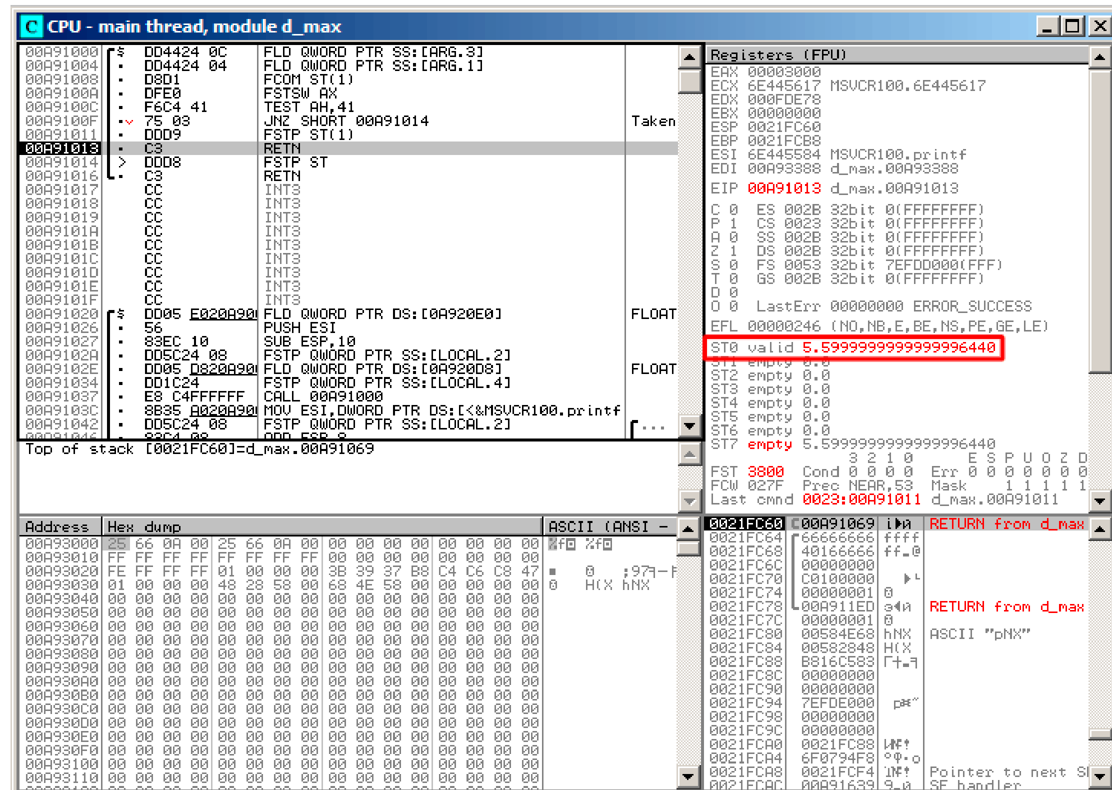


Abbildung 1.86: OllyDbg: FSTP wurde ausgeführt

Wir erkennen, dass der Befehl FSTP ST(1) wie folgt funktioniert: er lässt das oberste Element des Stacks an seinem Platz, löscht aber das Register ST(1).

GCC 4.4.1

Listing 1.189: GCC 4.4.1

```
d_max proc near

b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
    mov     ebp, esp
```

```
    sub     esp, 10h

; lege a und b auf den lokalen Stack:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; lade a und b auf den FPU Stack:

    fld     [ebp+a]
    fld     [ebp+b]

; aktueller Stand des Stacks: ST(0) - b; ST(1) - a

    fxch    st(1) ; dieser Befehl vertauscht ST(1) und ST(0)

; aktueller Stand des Stacks: ST(0) - a; ST(1) - b

    fucompp    ; vergleichee a und b und nimm zwei Werte (d.h. a und b) vom
    ; Stack
    fnstsw  ax ; speichere FPU Status in AX
    sahf       ; lade SF, ZF, AF, PF, und CF Flags aus AH
    setnbe  al ; speichere 1 in AL, falls CF=0 und ZF=0
    test    al, al             ; AL==0 ?
    jz      short loc_8048453 ; ja
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

FUCOMPP ist fast wie like FCOM, nimmt aber beide Werte vom Stand und behandelt „undefinierte Zahlenwerte“ anders.

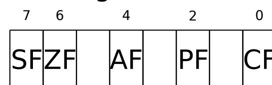
Ein wenig über *undefinierte Zahlenwerte*.

Die FPU ist in der Lage mit speziellen undefinieten Werten, den sogenannten *not-a-number*(kurz NaN) umzugehen. Beispiele sind etwa der Wert unendlich, das Ergebnis einer Division durch 0, etc. Undefinierte Werte können entwder „quiet“ oder „signaling“ sein. Es ist möglich mit „quiet“ NaNs zu arbeiten, aber beim Versuch einen Befehl auf „signaling“ NaNs auszuführen, wird eine Exception geworfen.

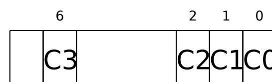
FCOM erzeugt eine Exception, falls irgendein Operand ein NaN ist. FUCOM erzeugt eine

Exception nur dann, wenn ein Operand eine „signaling“ NaN (SNaN) ist.

Der nächste Befehl ist SAHF (*Store AH into Flags*) —es handelt sich hierbei um einen seltenen Befehl, der nicht mit der FPU zusammenhängt. 8 Bits aus AH werden in die niederen 8 Bit der CPU Flags in der folgenden Reihenfolge verschoben:



Erinnern wir uns, dass FNSTSW die für uns interessanten Bits (C3/C2/C0) auf den Stellen 6,2,0 im AH Register setzt:



Mit anderen Worten: der Befehl `fnstsw ax / sahf` verschiebt C3/C2/C0 nach ZF, PF und CF.

Überlegen wir uns auch die Werte der C3/C2/C0 in unterschiedlichen Szenarien:

- Falls in unserem Beispiel a größer als b ist, dann werden die C3/C2/C0 auf 0,0,0 gesetzt.
- Falls a kleiner als b ist, werden die Bits auf 0,0,1 gesetzt.
- Falls $a = b$, dann werden die Bits auf 1,0,0 gesetzt.

Mit anderen Worten, die folgenden Zustände der CPU Flags sind nach drei FUCOMPP/FNSTSW/SAHF Befehlen möglich:

- Falls $a > b$, werden die CPU Flags wie folgt gesetzt `ZF=0, PF=0, CF=0`.
- Falls $a < b$, werden die CPU Flags wie folgt gesetzt: `ZF=0, PF=0, CF=1`.
- Und falls $a = b$, dann gilt: `ZF=1, PF=0, CF=0`.

Abhängig von den CPU Flags und Bedingungen, speichert SETNBE entweder 1 oder 0 in AL. Es ist also quasi das Gegenstück von JNBE mit dem Unterschied, dass SETcc

Depending on the CPU flags and conditions, SETNBE stores 1 or 0 to AL. It is almost the counterpart of JNBE, with the exception that SETcc[111] eine 1 oder 0 in AL speichert, aber Jcc tatsächlich auch springt. SETNBE speicher 1 nur, falls CF=0 und ZF=0. Wenn dies nicht der Fall ist, dann wird 0 in AL gespeichert.

Nur in einem Fall sind CF und ZF beide 0: falls $a > b$.

In diesem Fall wird 1 in AL gespeichert, der nachfolgende JZ Sprung wird nicht ausgeführt und die Funktion liefert _a zurück. In allen anderen Fällen wird _b zurückgegeben.

Optimierender GCC 4.4.1

Listing 1.190: Optimierender GCC 4.4.1

```
                public d_max
```

[111] *cc* is *condition code*

```
d_max           proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; Zustand des Stacks: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; Zustand des Stacks: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; vergleiche _a und _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

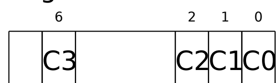
; speichere ST(0) in ST(0) (Befehl ohne Auswirkung),
; nimm obersten Wert vom Stack,
; lasse _b obenauf
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; speichere _a in ST(1), nimm obersten Wert vom Stand, lasse _a obenauf
                fstp    st(1)

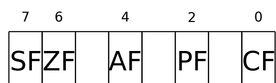
loc_804844A:
                pop     ebp
                retn
d_max           endp
```

Dies ist fast das gleiche, außer dass JA nach SAHF verwendet wird. Tatsächlich prüfen die bedingte Sprungbefehle, die vorzeichenlose Zahlen auf „größer“, „kleiner“ oder „gleich“ prüfen (das sind JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) lediglich die Flags CF und ZF.

Erinnern wir uns, an welchen Stellen die C3/C2/C0 sich im AH Register befinden, nachdem der Befehl FSTSW/FNSTSW ausgeführt wurde:



Halten wir uns auch vor Augen wie die Bits aus AH in den CPU Flags nach der Ausführung von SAHF abgelegt werden:



Nach dem Vergleich werden die C3 und C0 Bits nach ZF und CF verschoben, sodass der bedingte Sprung danach funktionieren kann. ird ausgeüführt, falls sowohl CF als

auch ZF gleich 0 sind.

Hierbei können alle hier aufgelisteten Sprungbefehle nach einem FNSTSW/SAHF Befehlspaar verwendet werden.

Offenbar wurden die C3/C2/C0 Status Bits der CPU dort bewusst platziert, sodass diese leicht auf die CPU Flags übertragen werden können, ohne dass zusätzliche Vertauschungen notwendig sind.

GCC 4.8.1 mit aktivierter -O3 Optimierung

Mit der P6 Intel Familie[112] wurden einige neue FPU Befehle hinzugefügt. Diese sind FUCOMI (vergleiche Operanden und setze Flags der CPU) und FCMOVcc (arbeitet wie CMOVcc, aber auf FPU Registern). Offenbar haben sich die Verwalter von GCC dazu entschieden, den Support von vor-P6 Intel CPUs (frühe Pentiums, 80486, etc.) einzustellen.

Außerdem ist die FPU nicht länger eine separate Einheit in der P6 Intel Familie, sodass es nun auch möglich ist, die Flags der CPU von der FPU aus zu prüfen oder zu verändern.

Wir erhalten also das Folgende:

Listing 1.191: Optimierender GCC 4.8.1

```
        fld     QWORD PTR [esp+4]       ; lade "a"
        fld     QWORD PTR [esp+12]      ; lade "b"
        ; ST0=b, ST1=a
        fxch    st(1)
        ; ST0=a, ST1=b
        ; vergleiche "a" und "b"
        fucomi  st, st(1)
        ; kopiere ST1 (hier: "b") nach ST0, falls a<=b
        ; lasse "a" sonst in ST0
        fcmovbe st, st(1)
        ; verwirf den Wert in ST1
        fstp    st(1)
        ret
```

Schwer zu sagen, warum FXCH (vertausche Operanden) hier verwendet wird.

Es ist möglich, diesen Befehl loszuwerden, indem man die ersten beiden FLD Befehle vertauscht oder FCMOVBE (*below or equal*) durch FCMOVA (*above*) ersetzt. Wahrscheinlich handelt es sich hierbei um eine Ungenauigkeit im Compiler.

FUCOMI vergleicht also ST(0) (a) und ST(1) (b) und setzt einige Flags in der CPU. FCMOVBE prüft die Flags und kopiert ST(1) (in diesem Moment also b) nach ST(0) (hier: a), falls $ST0(a) <= ST1(b)$. Andernfalls ($a > b$) wird a in ST(0) belassen.

Der letzte FSTP Befehl belässt ST(0) oben auf dem Stack und verwirft den Inhalt von ST(1).

Verfolgen wir den Funktionsverlauf in GDB:

[112] Beginnend mit Pentium Pro, Pentium-II, etc.

Listing 1.192: Optimierender GCC 4.8.1 and GDB

```
dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
dennis@ubuntuvm:~/polygon$ gdb d_max
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols ↙
    ↘ found)...done.
(gdb) b d_max
Breakpoint 1 at 0x80484a0
(gdb) run
Starting program: /home/dennis/polygon/d_max

Breakpoint 1, 0x080484a0 in d_max ()
(gdb) ni
0x080484a4 in d_max ()
(gdb) disas $eip
Dump of assembler code for function d_max:
   0x080484a0 <+0>:     fldl   0x4(%esp)
=> 0x080484a4 <+4>:     fldl   0xc(%esp)
   0x080484a8 <+8>:     fxch   %st(1)
   0x080484aa <+10>:    fucomi %st(1),%st
   0x080484ac <+12>:    fcmovbe %st(1),%st
   0x080484ae <+14>:    fstp   %st(1)
   0x080484b0 <+16>:    ret
End of assembler dump.
(gdb) ni
0x080484a8 in d_max ()
(gdb) info float
  R7: Valid   0x3fff9999999999999800 +1.199999999999999956
=>R6: Valid   0x4000d999999999999800 +3.399999999999999911
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a4
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) ni
0x080484aa in d_max ()
(gdb) info float
  R7: Valid   0x4000d999999999999800 +3.399999999999999911
=>R6: Valid   0x3fff9999999999999800 +1.199999999999999956
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
```

```
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
  R0: Empty   0x00000000000000000000

Status Word:         0x3000
                       TOP: 6
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0x0fff
Instruction Pointer: 0x73:0x080484a8
Operand Pointer:     0x7b:0xbffff118
Opcode:              0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
   0x080484a0 <+0>:     fldl   0x4(%esp)
   0x080484a4 <+4>:     fldl   0xc(%esp)
   0x080484a8 <+8>:     fxch   %st(1)
=> 0x080484aa <+10>:    fucomi %st(1),%st
   0x080484ac <+12>:    fcmovbe %st(1),%st
   0x080484ae <+14>:    fstp   %st(1)
   0x080484b0 <+16>:    ret
End of assembler dump.
(gdb) ni
0x080484ac in d_max ()
(gdb) info registers
eax            0x1      1
ecx            0xbffff1c4       -1073745468
edx            0x8048340        134513472
ebx            0xb7fbf000       -1208225792
esp            0xbffff10c       0xbffff10c
ebp            0xbffff128       0xbffff128
esi            0x0      0
edi            0x0      0
eip            0x80484ac        0x80484ac <d_max+12>
eflags         0x203    [ CF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) ni
0x080484ae in d_max ()
(gdb) info float
  R7: Valid   0x4000d999999999999800 +3.399999999999999911
=>R6: Valid   0x4000d999999999999800 +3.399999999999999911
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
```

```
105   R0: Empty   0x00000000000000000000
106
107 Status Word:         0x3000
108                        TOP: 6
109 Control Word:        0x037f   IM DM ZM OM UM PM
110                        PC: Extended Precision (64-bits)
111                        RC: Round to nearest
112 Tag Word:            0x0fff
113 Instruction Pointer: 0x73:0x080484ac
114 Operand Pointer:     0x7b:0xbffff118
115 Opcode:              0x0000
116 (gdb) disas $eip
117 Dump of assembler code for function d_max:
118    0x080484a0 <+0>:     fldl   0x4(%esp)
119    0x080484a4 <+4>:     fldl   0xc(%esp)
120    0x080484a8 <+8>:     fxch   %st(1)
121    0x080484aa <+10>:    fucomi %st(1),%st
122    0x080484ac <+12>:    fcmovbe %st(1),%st
123 => 0x080484ae <+14>:    fstp   %st(1)
124    0x080484b0 <+16>:    ret
125 End of assembler dump.
126 (gdb) ni
127 0x080484b0 in d_max ()
128 (gdb) info float
129 =>R7: Valid   0x4000d999999999999800 +3.399999999999999911
130   R6: Empty   0x4000d999999999999800
131   R5: Empty   0x00000000000000000000
132   R4: Empty   0x00000000000000000000
133   R3: Empty   0x00000000000000000000
134   R2: Empty   0x00000000000000000000
135   R1: Empty   0x00000000000000000000
136   R0: Empty   0x00000000000000000000
137
138 Status Word:         0x3800
139                        TOP: 7
140 Control Word:        0x037f   IM DM ZM OM UM PM
141                        PC: Extended Precision (64-bits)
142                        RC: Round to nearest
143 Tag Word:            0x3fff
144 Instruction Pointer: 0x73:0x080484ae
145 Operand Pointer:     0x7b:0xbffff118
146 Opcode:              0x0000
147 (gdb) quit
148 A debugging session is active.
149
150         Inferior 1 [process 30194] will be killed.
151
152 Quit anyway? (y or n) y
153 dennis@ubuntuvm:~/polygon$
```

Unter Verwendung von „ni“ führen wir die ersten beiden FLD Befehle aus.

Sehen wir uns die FPU Register (Zeile 33) an.

Wie bereits erwähnt, bildet der FPU Registersatz einen Ringpuffer und keinen Stack (1.19.5 on page 261). Außerdem zeigt GDB nicht die STx Register, sondern die internen FPU Register (Rx). Der Pfeil (in Zeile 35) zeigt auf das aktuell obere Ende des Stacks.

Wir sehen auch den Inhalt des TOP Registers in *Status Word* (Zeile 36-37)–hier ist dieser 6, sodass das oberste Element im Stack also aktuell auf das interne Register 6 zeigt.

Die Werte von a und b werden nach Ausführung von FXCH (Zeile 54) vertauscht.

FUCOMI wird ausgeführt (Zeile 83). Betrachten wir die Flags: CF ist gesetzt (Zeile 95).

FCMOVBE hat den Wert von b kopiert (siehe Zeile 104).

FSTP lässt einen Wert oben auf dem Stack (Zeile 139). Der Wert von TOP beträgt jetzt 7, was bedeutet, dass das obere Ende des FPU Stacks jetzt auf das interne Register 7 zeigt.

ARM

Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Listing 1.193: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; kopiere "a" nach D16
VMOV            R0, R1, D16
BX              LR
```

Ein recht einfacher Fall. Die Eingabewerte werden in die Register D17 und D16 geladen und dann mit dem Befehl VCMPE verglichen.

Genau wie der x86-Koprozessor besitzt auch der ARM-Koprozessor seine eigenen Status und Flag Register (FPSCR[113]), denn es gibt auch hier die Notwendigkeit die spezifischen Flags des Koprozessors zu speichern.

Und genau wie beim x86 gibt es auch in ARM keine Befehle für bedingte Sprünge, die die Bits im Statusregister des Koprozessors abfragen können. So gibt es den Befehl VMRS, um 4 Bits (N, Z, C, V) vom Statusregister des Koprozessors in das *allgemeine* Statusregister (APSR[114]) zu kopieren.

VMOVGT ist das Analogon zum MOVGT Befehl für D-Register: er wird ausgeführt, wenn ein Operand bezüglich eines *GT—Greater Than (dt. größer als)* Vergleichs größer ist als der andere.

Wenn er ausgeführt wird, wird der Wert von a nach D16 geschrieben (dieser wird aktuell in D17 gespeichert). Andernfalls bleibt der Wert von b im D16 Register.

[113](ARM) Floating-Point Status and Control Register
[114](ARM) Application Program Status Register

Der vorletzte Befehl VMOV bereitet den Wert im D16 Register für die Rückgabe über das Registerpaar R0 und R1 vor.

Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

Listing 1.194: Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Fast das gleiche wie im vorherigen Beispiel, aber in gewisser Weise dennoch unterschiedlich. Wie wir bereits wissen, können viele Befehl im ARM mode durch bedingte Prädikate unterstützt werden. Im Thumb mode dagegen gibt es nichts Vergleichbares. Es gibt keinen Platz in den 16-Bit-Befehlen für 4 weitere Bits, in denen Bedingungen kodiert werden könnten.

Thumb-2 wurde erweitert, um zu ermöglichen alten Thumb-Befehlen nachträglich Prädikate zuzuweisen. Hier, im von IDA erzeugten Listing finden wir den VMOVGT Befehl aus dem vorherigen Beispiel wieder.

Tatsächlich ist hier das gewöhnliche VMOV kodiert, aber IDA ergänzt den Suffix -GT, da sich direkt davor eine IT GT Befehl befindet.

Der IT Befehl definiert einen sogenannten *If-Then-Block*.

Nach dem Befehl können bis zu 4 weitere Befehle, jeder mit einem beschreibenden Suffix, verwendet werden. In unserem Beispiel impliziert IT GT, dass der Folgebefehl genau dann ausgeführt werden soll, wenn die ITGT (*Greater Than*) Bedingung wahr ist.

Hier ist ein komplexeres Codefragment, welches aus Angry Birds (für iOS) stammt:

Listing 1.195: Angry Birds Classic

```
...
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
BLX             _objc_msgSend ; ohne Suffix
...
```

ITE steht für *if-then-else* und kodiert Suffixe für die beiden folgenden Befehle.

Der erste Befehl wird ausgeführt, wenn die durch ITE (*NE, not ewual*, dt. ungleich) kodierte Bedingung wahr ist und der zweite wenn die Bedingung falsch ist (die inverse Bedingung zu NE ist EQ (*equal*, dt. gleich)).

Der dem zweiten Befehl folgende VMOV (oder VMOVEQ) ist ein gewöhnlicher Befehl ohne Suffix (BLX).

Ein weiteres etwas schwieriger verständliches Codefragment, ebenfalls aus Angry Birds:

Listing 1.196: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
BLX             ___stack_chk_fail ; ohne Suffix
...
```

Vier „T" Symbole in der Mnemonik des Befehls bedeuten, dass die vier folgenden Befehle alle ausgeführt werden, wenn die Bedingung wahr ist.

Aus diesem Grund fügt IDA den -EQ Suffix an jeden der vier Befehle an.

Gäbe es beispielsweise ITEEE EQ (*if-then-else-else-else*), dann würden wie folgt Suffixe angehängt werden:

```
-EQ
-NE
-NE
-NE
```

Ein weiteres Fragment aus Angry Birds:

Listing 1.197: Angry Birds Classic

```
...
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
MOVS            R6, #0         ; ohne Suffix
CBZ             R0, loc_1E7E32 ; ohne Suffix
...
```

ITTE (*if-then-then-else*) impliziert, dass der erste und zweite Befehl ausgeführt werden, wenn die LE (*Less or Equal*, dt. mindestens) Bedingung wahr ist und der dritte, wenn die inverse Bedingung (GT—*Greater Than*, dt. mehr als) wahr ist.

Für gewöhnlich erzeugen Compiler nicht alle denkbaren Kombinationen. Im betrachteten Spiel Angry Birds beispielsweise (*classic* Version für iOS) werden nur die folgenden Variationen des IT Befehls verwendet: IT, ITE, ITT, ITTE, ITTT, ITTTT. Bleibt die Frage, wie man dies lernen kann. In IDA ist es mögliche Listing-Dateien zu erzeugen mit der Option 4 Bytes für jeden Opcode anzuzeigen. Dadurch können wir bei Kenntnis des höherwertigen Teils des 16-Bit-Opcodes (IT entspricht 0xBF) unter Zuhilfenahme von grep wie folgt vorgehen:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

Übrigens, wenn man von Hand Assemblerprogramme für ARM in Thumb-2 mode schreibt und man die Suffixe für die Bedingungen selbst anhängt, wird der Assemblierer die entsprechenden IT Befehle inklusive der benötigten Flags automatisch an den benötigten Stellen hinzufügen.

Nicht optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Listing 1.198: Nicht optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Fast identisch mit dem, was wir schon gesehen haben, aber hier gibt es zu viel redundanten Code, weil die Variablen a und b im lokalen Stack und zusätzlich als Rückgabewerte gespeichert werden.

Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.199: Optimierender Keil 6/2013 (Thumb Modus)

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
                MOVS    R5, R3
                MOVS    R6, R0
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil erzeugt keine FPU-Befehle, da er sich sich darauf verlassen kann, dass diese von der Ziel-CPU unterstützt werden und dies nicht durch einfache bitweisen Vergleich erledigt werden kann.

Keil ruft also eine Funktion einer externen Programmbibliothek (__aeabi_cdrcmple) auf, um den Vergleich durchzuführen. Das Ergebnis des Vergleich wird von der Funktion in den Flags belassen, sodass der folgende BCS (*Carry set—Greater than or equal*) Befehl ohne zusätzlichen Code funktioniert.

ARM64

Optimierender GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
        fcmpe   d0, d1
        fcsel   d0, d0, d1, gt
; Ergebnis jetzt in D0
        ret
```

Der ARM64 ISA verfügt über FPU-Befehle, die der Einfachheit halber die Flags der CPU APSR anstelle von FPSCR setzen. Die FPU ist hier kein separates Gerät mehr (zumindest logisch).

Wir finden hier FCMPE. Er vergleicht die beiden über D0 und D1 übergebenen Werte (dabei handelt es sich um das erste und zweite Argument der Funktion) und setzt APSR die Flags (N, Z, C, V).

FCSEL (*Floating Conditional Select*) kopiert den Wert von D0 oder D1 nach D0, abhängig von der Bedingung (GT—Greater Than), und verwendet wiederum Flags im APSR Register anstatt derer von FPSCR.

Dies ist im Vergleich zum Befehlssatz alter CPUs deutlich praktischer.

Falls die Bedingung wahr ist (GT), dann wird der Wert von D0 nach D0 kopiert (d.h. es geschieht nichts). Falls die Bedingung falsch ist, wird der Wert von D1 nach D0 kopiert.

Nicht optimierender GCC (Linaro) 4.9

```
d_max:
; speichere Eingabeparameter in der "Register Save Area"
        sub     sp, sp, #16
        str     d0, [sp,8]
        str     d1, [sp]
; lade Werte erneut
        ldr     x1, [sp,8]
        ldr     x0, [sp]
        fmov    d0, x1
        fmov    d1, x0
; D0 - a, D1 - b
        fcmpe   d0, d1
        ble     .L76
; a>b; lade D0 (a) nach X0
        ldr     x0, [sp,8]
        b       .L74
.L76:
; a<=b; lade D1 (b) nach X0
        ldr     x0, [sp]
.L74:
; result in X0
        fmov    d0, x0
; result in D0
        add     sp, sp, 16
        ret
```

Der nicht optimierende GCC ist weniger kompakt.

Zunächst speichert die Funktion ihre Eingabewerte auf dem lokalen Stack (*Register Save Area*), danach lädt der Code die Werte erneut in die Register X0/X1 und kopiert sie schließlich nach D0/D1, um sie mittels FCMPE zu vergleichen. Eine Menge redundanter Code, aber so arbeitet ein nicht optimierender Compiler nun einmal. FCMPE vergleich die Werte und setzt die APSR Flags. Zu diesem Zeitpunkt entscheidet sich der Compiler noch nicht für den praktischeren FCSEL Befehl und arbeitet stattdessen mit herkömmlichen Methoden: er verwendet den BLE Befehl (*Branch if Less than or Equal*). Im ersten Fall ($a > b$) wird der Wert von a nach X0 geladen. Im anderen Fall ($a <= b$) wird der Wert von b nach X0 geladen. Schließlich wird der Wert aus X0 nach D0 kopiert, denn der Rückgabewert muss sich in diesem Register befinden.

Übung

Dem Leser bleibt als Übung, den vorstehenden Code zu optimieren, indem manuell die redundanten Instruktionen entfernt werden ohne dabei neue einzuführen (wie etwa FCSEL).

Optimierender GCC (Linaro) 4.9—float

Wir wollen nun dieses Beispiel umschreiben, indem wir *float* anstelle von *double* verwenden.

```
float f_max (float a, float b)
{
        if (a>b)
                return a;

        return b;
};
```

```
f_max:
; S0 - a, S1 - b
        fcmpe   s0, s1
        fcsel   s0, s0, s1, gt
; Ergebnis jetzt in S0
        ret
```

Es ist der gleiche Code, aber hier werden die S-Register anstelle der D-Register verwendet. Das ist darauf zurückzuführen, dass der *float* Typ in 32-Bit-S-Registern übergeben wird (welche in Wirklichkeit nichts anderes als die niederen Teile der 64-Bit-D-Register sind).

MIPS

Der Koprozessor des MIPS Prozessors hat ein Condition Bit, welches in der FPU gesetzt und in der CPU geprüft werden kann.

Frühere MIPS haben nur ein Condition Bit (genannt FCC0), spätere haben deren 8 (genannt FCC7-FCC0).

Diese(s) Bit(s) befinden sich im Register FCCR.

Listing 1.200: Optimierender GCC 4.4.5 (IDA)

```
d_max:
; setze das FPU Condition Bit, falls $f14<$f12 (b<a):
                c.lt.d  $f14, $f12
                or      $at, $zero ; NOP
; springe zu locret_14 , falls das Condition Bit gesetzt ist
                bc1t    locret_14
; dieser Befehl wird stets ausgeführt (setze Rückgabewert auf "a"):
                mov.d   $f0, $f12  ; branch delay slot
; dieser Befehl wird nur ausgeführt, falls der Zweig nicht betreten wurde
; (d.h., falls b>=a)
; setze Rückgabewert auf "b":
                mov.d   $f0, $f14

locret_14:
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

C.LT.D vergleicht zwei Werte. LT ist die Bedingung „Less Than" (weniger als). D impliziert einen Wert vom Typ *double*. Abhängig vom Ergebnis des Vergleichs wird das FCC0 Condition Bit entweder gesetzt oder gelöscht.

BC1T prüft das FCC0 Bit und sprint, falls das Bit gesetzt ist. T bedeutet, dass der Sprung ausgeführt wird, wenn das Bit gesetzt („True") ist. Daneben gibt es auch den Befehl BC1F, der springt, wenn das Bit gelöscht („FALSE") ist.

Abhängig vom Sprung wird einer der Funktionsargument in $F0 abgelegt.

1.19.8 Einige Konstanten

Es ist leicht, in Wikipedia die Darstellungen einiger Konstanten nach dem IEEE 754 Standard nachzulesen. Es ist interessant zu wissen, dass 0.0 nach IEEE 754 als 32 Nullbits (in einfacher Genauigkeit) oder 64 Nullbits (in doppelter Genauigkeit) dargestellt wird. Wenn also eine Fließkommavariable im Register oder Speicher auf 0.0 gesetzt werden soll, kann der Befehl MOV oder *XOR reg, reg* verwendet werden. Dies ist geeignet für Strukturen, in denen viele Variable unterschiedlichster Datentypen vorhanden sind. Mit der gewöhnlichen memset() Funktion kann man alle Integervariablen auf 0, alle Booleschen Variablen auf *false*, alle Pointer auf NULL und alle Fließkommavariablen (beliebiger Genauigkeit) auf 0.0 setzen.

1.19.9 Kopieren

Man könnte fälschlicherweise annehmen, dass die Befehle FLD/FST verwendet werden müssen, um IEEE 754 Werte zu laden oder zu speichern (also auch zu kopieren). Nichtsdestotrotz kann dies einfacher durch den Befehl MOV erreicht werden, welcher, logischerweise, Werte bitweise kopiert.

1.19.10 Stack, Taschenrechner und umgekehrte polnische Notation

Jetzt können wir einsehen, warum manche alten Taschenrechner die umgekehrte polnische Notation verwenden.

Für die Addition von 12 und 34 muss beispielsweise zuerst 12, dann 34 und dann das „plus"-Zeichen eingegeben werden.

Dies liegt daran, dass alte Taschenrechner als einfache Stackmaschinen implementiert waren und es auf diese Weise wesentlich einfacher war, mit komplexen geklammerten Ausdrücke umzugehen.

1.19.11 x64

Mehr dazu wie Fließkommazahlen in x86-64 verarbeitet werden unter: 1.29 on page 510

1.19.12 Übungen

- http://challenges.re/60
- http://challenges.re/61

1.20 Arrays

yy Ein Array ist nur ein Satz Variablen vom gleichen Typ die im Speicher aufeinander folgen[115]

1.20.1

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

x86

MSVC

Kompilieren wir das Beispiel:

Listing 1.201: MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84                        ; size = 4
_a$ = -80                        ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84        ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
```

[115] AKA „homogener Container“.

```
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20    ; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20     ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _a$[ebp+ecx*4]
    push    edx
    mov     eax, DWORD PTR _i$[ebp]
    push    eax
    push    OFFSET $SG2463
    call    _printf
    add     esp, 12         ; 0000000cH
    jmp     SHORT $LN2@main
$LN1@main:
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

Soweit nichts Außergewöhnliches, nur zwei Schleifen: die erste füllt mit Werten auf und die zweite gibt Werte aus. Der Befehl `shl ecx, 1` wird für die Multiplikation mit 2 in ECX verwendet; mehr dazu unten 1.18.2 on page 253.

Auf dem Stack werden 80 Bytes für das Array reserviert: 20 Elemente von je 4 Byte.

Untersuchen wir dieses Beispiel in OllyDbg.

Wir erkennen wie das Array befüllt wird:

jedes Element ist ein 32-Bit-Wort vom Typ *int* und der Wert ist der Index multipliziert mit 2:

Abbildung 1.87: OllyDbg: nach dem Füllen des Arrays

Da sich dieses Array auf dem Stack befindet, finden wir dort alle seine 20 Elemente.

GCC

Hier ist was GCC 4.4.1 erzeugt:

Listing 1.202: GCC 4.4.1

```
                public main
main            proc near                   ; DATA XREF: _start+17
```

```
var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1          ; i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

Die Variable a ist übrigens vom Typ *int** (Pointer auf *int*)–man kann einen Pointer auf ein Array an eine andere Funktion übergeben, aber es ist richtiger zu sagen, dass der Pointer auf das erste Element des Arrays übergeben wird. (Die Adressen der übrigen Elemente werden in bekannter Weise berechnet.)

Wenn man diesen Pointer mittels *a[idx]* indiziert, wird *idx* zum Pointer addiert und das dort abgelegte Element (auf das der berechnete Pointer zeigt) wird zurückgegeben.

Ein interessantes Beispiel: ein String wie *string* ist ein Array von Chars und hat den Typ *const char[]*.

Auch auf diesen Pointer kann ein Index angewendet werden.

Das ist der Grund warum es es möglich ist, Dinge wie „`string`“`[i]` zu schreiben–es handelt sich dabei um einen korrekten C/C++ Ausdruck!

ARM

Nicht optimierender Keil 6/2013 (ARM Modus)

```
            EXPORT _main
_main
            STMFD   SP!, {R4,LR}
            SUB     SP, SP, #0x50    ; Platz für 20 int Variablen reservieren

; erste Schleife

            MOV     R4, #0           ; i
            B       loc_4A0
loc_494
            MOV     R0, R4,LSL#1     ; R0=R4*2
            STR     R0, [SP,R4,LSL#2]; sichere R0 in SP+R4<<2 (entspricht
    SP+R4*4)
            ADD     R4, R4, #1       ; i=i+1

loc_4A0
            CMP     R4, #20          ; i<20?
            BLT     loc_494          ; falls ja, Rumpf der Schleife erneut
    ausführen

; zweite Schleife

            MOV     R4, #0           ; i
            B       loc_4C4
loc_4B0
            LDR     R2, [SP,R4,LSL#2]; (zweites printf Argument)
    R2=*(SP+R4<<4) (entspricht *(SP+R4*4))
            MOV     R1, R4           ; (erstes printf Argument) R1=i
            ADR     R0, aADD         ; "a[%d]=%d\n"
            BL      __2printf
            ADD     R4, R4, #1       ; i=i+1

loc_4C4
            CMP     R4, #20          ; i<20?
            BLT     loc_4B0          ; falls ja, Rumpf der Schleife erneut
    ausführen
            MOV     R0, #0           ; Rückgabewert
            ADD     SP, SP, #0x50    ; Block freigeben, der für die 20 int
    Variablen reserviert wurde
            LDMFD   SP!, {R4,PC}
```

Der Typ *int* benötigt 32 Bit (oder 4 Byte) zum Speichern, weshalb zum Speichern von

20 *int* Variablen 80 (0x50) Bytes benötigt werden. Deshalb reserviert der Befehl SUB SP, SP, #0x50 im Funktionsprolog genau diese Menge an Speicherplatz auf dem Stack.

In sowohl der ersten als auch der zweiten Scheife befindet sich der Scheifenzähler *i* im R4 Register.

Die Zahl, die in das Array geschrieben wird, wird über den Ausdruck $i \cdot 2$ berechnet, was äquivalent zur Linksverschiebung um 1 Bit ist, weshalb der Befehl MOV R0, R4,LSL#1 verwendet wird.

STR R0, [SP,R4,LSL#2] schreibt den Inhalt von R0 in das Array.

Ein Pointer auf ein Arrayelement wird wie folgt berechnet: **SP!** zeigt auf den Beginn des Arrays, Reg4 ist i. Eine Linksverschiebung von i um 2 Bits entspricht effektiv einer Multiplikation mit 4 (da jedes Arrayelement eine Größe von 4 Byte hat) und wird dann zur Adresse am Beginn des Arrays addiert.

Die zweite Schleife enthält den inversen Befehl LDR R2, [SP,R4,LSL#2]. Er lädt den benötigten Wert aus dem Array und der Pointer hierauf wird analog berechnet.

Optimierender Keil 6/2013 (Thumb Modus)

```
_main
          PUSH    {R4,R5,LR}
; Platz für 20 int Variablen und eine weitere Variable reservieren
          SUB     SP, SP, #0x54

; erste Schleife

          MOVS    R0, #0      ; i
          MOV     R5, SP      ; Pointer auf das erste Arrayelement

loc_1CE
          LSLS    R1, R0, #1  ; R1=i<<1 (entspricht i*2)
          LSLS    R2, R0, #2  ; R2=i<<2 (entspricht i*4)
          ADDS    R0, R0, #1  ; i=i+1
          CMP     R0, #20     ; i<20?
          STR     R1, [R5,R2] ; sichere R1 in *(R5+R2) (entspricht R5+i*4)
          BLT     loc_1CE     ; falls i<20, führe Schleifenrumpf erneut aus

; zweite Schleife

          MOVS    R4, #0      ; i=0
loc_1DC
          LSLS    R0, R4, #2  ; R0=i<<2 (entspricht i*4)
          LDR     R2, [R5,R0] ; lade von *(R5+R0) (entspricht R5+i*4)
          MOVS    R1, R4
          ADR     R0, aADD    ; "a[%d]=%d\n"
          BL      __2printf
          ADDS    R4, R4, #1  ; i=i+1
          CMP     R4, #20     ; i<20?
          BLT     loc_1DC     ; falls i<20, führe Schleifenrumpf erneut aus
```

```
          MOVS    R0, #0      ; Rückgabewert
; Block freigeben, der für die 20 int Variablen
; und die weitere Variable reserviert wurde
          ADD     SP, SP, #0x54
          POP     {R4,R5,PC}
```

Der Thumb Code ist sehr ähnlich. Der Thumb mode kennt spezielle Befehl für das bitweise Verschieben (wie LSLS), der den in das Array zu schreibenden Wert und die Adresse jedes Elements im Array berechnet.

Der Compiler reserviert ein wenig mehr Platz auf dem lokalen Stack, aber die letzten 4 Byte davon werden nicht verwendet.

Nicht optimierender GCC 4.9.1 (ARM64)

Listing 1.203: Nicht optimierender GCC 4.9.1 (ARM64)

```
.LC0:
        .string "a[%d]=%d\n"
main:
; speichere FP und LR im Stackframe:
        stp     x29, x30, [sp, -112]!
; setze Stackframe (FP=SP)
        add     x29, sp, 0
; initialisiere Zählervariable mit 0 (WZR ist das Register,
; das immer 0 enthält):
        str     wzr, [x29,108]
; springe zum Code, der die Schleifenbedingung prüft:
        b       .L2
.L3:
; lade Wert der Variablen "i":
        ldr     w0, [x29,108]
; multipliziere ihn mit 2:
        lsl     w2, w0, 1
; find a place of an array in local stack:
        add     x0, x29, 24
; lade 32-Bit-Integer vom lokalen Stack und erweitere ihn auf 64 Bit:
        ldrsw   x1, [x29,108]
; berechne Adresse des Elements (X0+X1<<2=Arrayadresse+i*4)
; und speichere W2 (i*2) dort:
        str     w2, [x0,x1,lsl 2]
; inkrementiere Zähler (i):
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L2:
; prüfen, ob wir fertig sind:
        ldr     w0, [x29,108]
        cmp     w0, 19
; springe zu L3 (Beginn des Schleifenrumpfes) falls nicht:
        ble     .L3
; der zweite Teil der Funktion fängt hier an.
```

```
; initialisiere Zählervariable mit 0
; es wurde übrigens derselbe Platz im lokalen Stack für den Zähler verwendet,
; da dieselbe lokale Variable (i) als Zähler verwendet wird.
        str     wzr, [x29,108]
        b       .L4
.L5:
; berechne Arrayadresse:
        add     x0, x29, 24
; lade "i" Wert:
        ldrsw   x1, [x29,108]
; lade Wert aus dem Array von der Adresse (X0+X1<<2 = Adresse des Arrays +
   i*4)
        ldr     w2, [x0,x1,lsl 2]
; lade Adresse des "a[%d]=%d\n" Strings:
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
; lade Variable "i" nach W1 und übergebe sie an printf() als zweites
   Argument:
        ldr     w1, [x29,108]
; W2 enthält immer noch den Wert des gerade geladenen Arrayelements.
; Aufruf von printf():
        bl      printf
; inkrementiere "i":
        ldr     w0, [x29,108]
        add     w0, w0, 1
        str     w0, [x29,108]
.L4:
; sind wir fertig?
        ldr     w0, [x29,108]
        cmp     w0, 19
; springe zum Schleifenrumpf, falls nicht:
        ble     .L5
; gib 0 zurück
        mov     w0, 0
; stelle FP und LR wieder her:
        ldp     x29, x30, [sp], 112
        ret
```

MIPS

Die Funktion verwendet eine Menge S-Register, die gesichert werden müssen. Das ist der Grund dafür, dass die Werte im Funktionsprolog gespeichert und im Funktionsepilog wiederhergestellt werden.

Listing 1.204: Optimierender GCC 4.4.5 (IDA)

```
main:

var_70          = -0x70
var_68          = -0x68
var_14          = -0x14
var_10          = -0x10
var_C           = -0xC
```

```
var_8           = -8
var_4           = -4
; Funktionsprolog:
                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x80
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x80+var_4($sp)
                sw      $s3, 0x80+var_8($sp)
                sw      $s2, 0x80+var_C($sp)
                sw      $s1, 0x80+var_10($sp)
                sw      $s0, 0x80+var_14($sp)
                sw      $gp, 0x80+var_70($sp)
                addiu   $s1, $sp, 0x80+var_68
                move    $v1, $s1
                move    $v0, $zero
; dieser Wert wird als terminierendes Zeichen für die Schleife verwendet.
; er wurde vom GCC Compiler zur Compilerzeit vorausberechnet
                li      $a0, 0x28  # '('

loc_34:                                  # CODE XREF: main+3C
; speichere Wert:
                sw      $v0, 0($v1)
; erhöhe zu speichernden Wert bei jeder Iteration um 2
                addiu   $v0, 2
; terminierendes Zeichen erreicht?
                bne     $v0, $a0, loc_34
; immer 4 zur Adresse addieren:
                addiu   $v1, 4
; Schleife zum Befüllen des Arrays ist beendet
; Beginn der zweiten Schleife
                la      $s3, $LC0        # "a[%d]=%d\n"
; Variable "i" bleibt in $s0:
                move    $s0, $zero
                li      $s2, 0x14

loc_54:                                  # CODE XREF: main+70
; Aufruf von printf():
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $a2, 0($s1)
                move    $a1, $s0
                move    $a0, $s3
                jalr    $t9
; erhöhe "i":
                addiu   $s0, 1
                lw      $gp, 0x80+var_70($sp)
; springe zum Rumpf der Schleife, falls das Ende noch nicht erreicht ist
                bne     $s0, $s2, loc_54
; setze Memory Pointer auf das nächste 32-Bit-Wort:
                addiu   $s1, 4
; Funktionsepilog
                lw      $ra, 0x80+var_4($sp)
                move    $v0, $zero
                lw      $s3, 0x80+var_8($sp)
```

```
                lw      $s2, 0x80+var_C($sp)
                lw      $s1, 0x80+var_10($sp)
                lw      $s0, 0x80+var_14($sp)
                jr      $ra
                addiu   $sp, 0x80

$LC0:           .ascii "a[%d]=%d\n"<0>   # DATA XREF: main+44
```

Interessant: es gibt zwei Schleifen und die erste benötigt i nicht; sie benötigt nur $i \cdot 2$ (erhöht um 2 bei jedem Iterationsschritt) und die Adresse im Speicher (erhöht um 4 bei jedem Iterationsschritt).

Wir sehen hier also zwei Variablen: eine (in $V0), die jedes Mal um 2 erhöht wird, und eine andere (in$V1), die um 4 erhöht wird.

Die zweite Schleife ist der Ort, an dem `printf()` aufgerufen wird und dem Benutzer den Wert von i zurückliefert, es gibt also eine Variable die in $S0 inkrementiert wird und eine Speicheradresse in $S1, die jedes Mal um 4 erhöht wird.

Das erinnert uns an die Optimierung von Schleifen, die wir früher betrachtet haben: **??** on page ??.

Das Ziel der Optimierung ist es, die Multiplikationen loszuwerden.

1.20.2 Puffer-Überlauf

Lesezugriff außerhalb von Arraygrenzen

Der indizierte Zugriff auf ein Array wird durch *array[index]* realisiert. Wenn man sich den erzeugten Code genau ansieht, bemerkt man, dass eine Prüfung der Indexgrenzen fehlt, welche die Bedingung *kleiner als 20* validiert. Was also passiert, wenn der Index 20 oder größer ist? Hier haben wir es mit einem unschönen Feature von C/C++ zu tun

Hier ein Beipsielcode der erfolgreich kompiliert wurde und funktioniert:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[20]=%d\n", a[20]);

        return 0;
};
```

Ergebnis des Kompiliervorgangs (MSVC 2008):

Listing 1.205: Nicht optimierender MSVC 2008

```
$SG2474 DB    'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN2@main
$LN1@main:
    mov    eax, DWORD PTR _a$[ebp+80]
    push   eax
    push   OFFSET $SG2474 ; 'a[20]=%d'
    call   DWORD PTR __imp__printf
    add    esp, 8
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Der Code produziert dieses Ergebnis:

Listing 1.206: OllyDbg: console output

```
a[20]=1638280
```

Es handelt sich um *irgendetwas*, das auf dem Stack in der Nähe des Arrays gelegen hat, 80 Byte von dessen erstem Element entfernt.

Versuchen wir mit OllyDbg herauszufinden, woher dieser Wert kommt.

Laden und finden wir also den Wert, der sich direkt hinter dem letzten Arrayelement befindet:

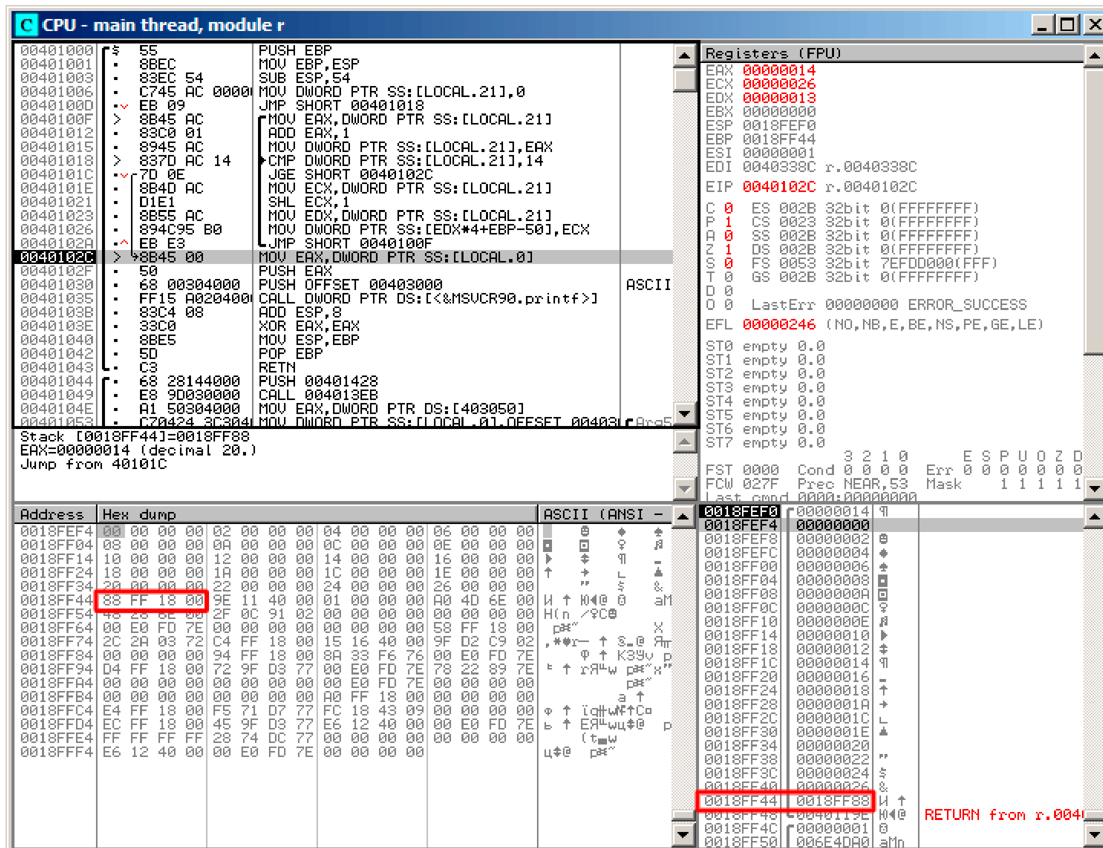


Abbildung 1.88: OllyDbg: das 20. Element lesen und `printf()` ausführen

Worum handelt es sich? Dem Stacklayout nach zu urteilen ist dies der gespeicherte Wert des EBP Registers.

Verfolgen wir das ganze weiter und schauen uns an, wie dieser wiederhergestellt wird:

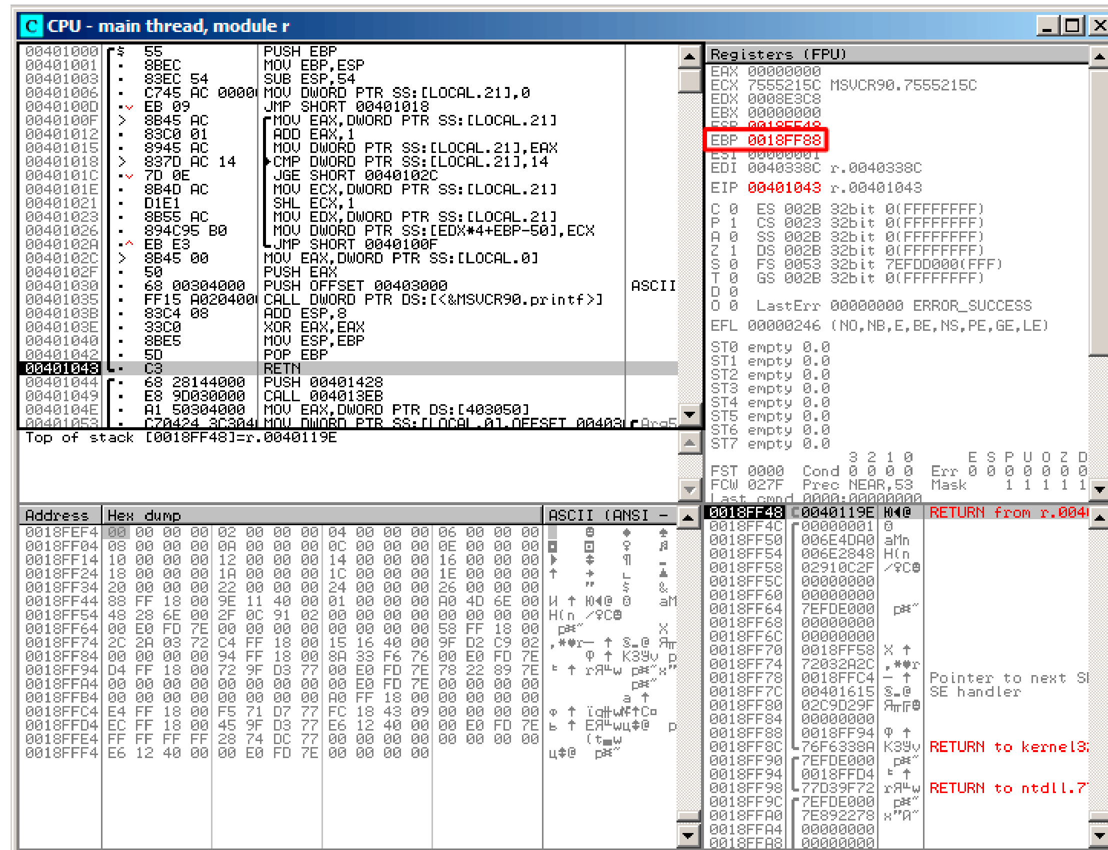


Abbildung 1.89: OllyDbg: Wert von EBP wiederherstellen

Wie könnte es anders gelöst werden? Der Compiler könnte zusätzlichen Code erzeugen, der sicherstellt, dass der Index sich stets innerhalb der Arraygrenzen befindet (wie in höheren Programmiersprachen[116]), aber das würde den Code langsamer machen.

Schreibzugriff außerhalb von Arraygrenzen

Nehmen wir an, wir hätte ein paar Werte illegalerweise vom Stack gelesen, wie könnten wir etwas hineinschreiben?

Hier ist, was wir haben:

```
#include <stdio.h>
```

[116]Java, Python, etc.

```
int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

MSVC

Wir erhalten das Folgende:

Listing 1.207: Nicht optimierender MSVC 2008

```
_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30 ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]      ; dieser Befehl ist offensichtlich
    redundant
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX könnte hier als zweiter Operand
    verwendet werden
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Das kompilierte Programm stürzt nach der Ausführung ab. Das verwundert nicht. Schauen wir, was genau den Absturz verursacht.

Laden wir das Programm in OllyDbg und verfolgen den Ablauf, bis alle 30 Elemente geschrieben worden sind:

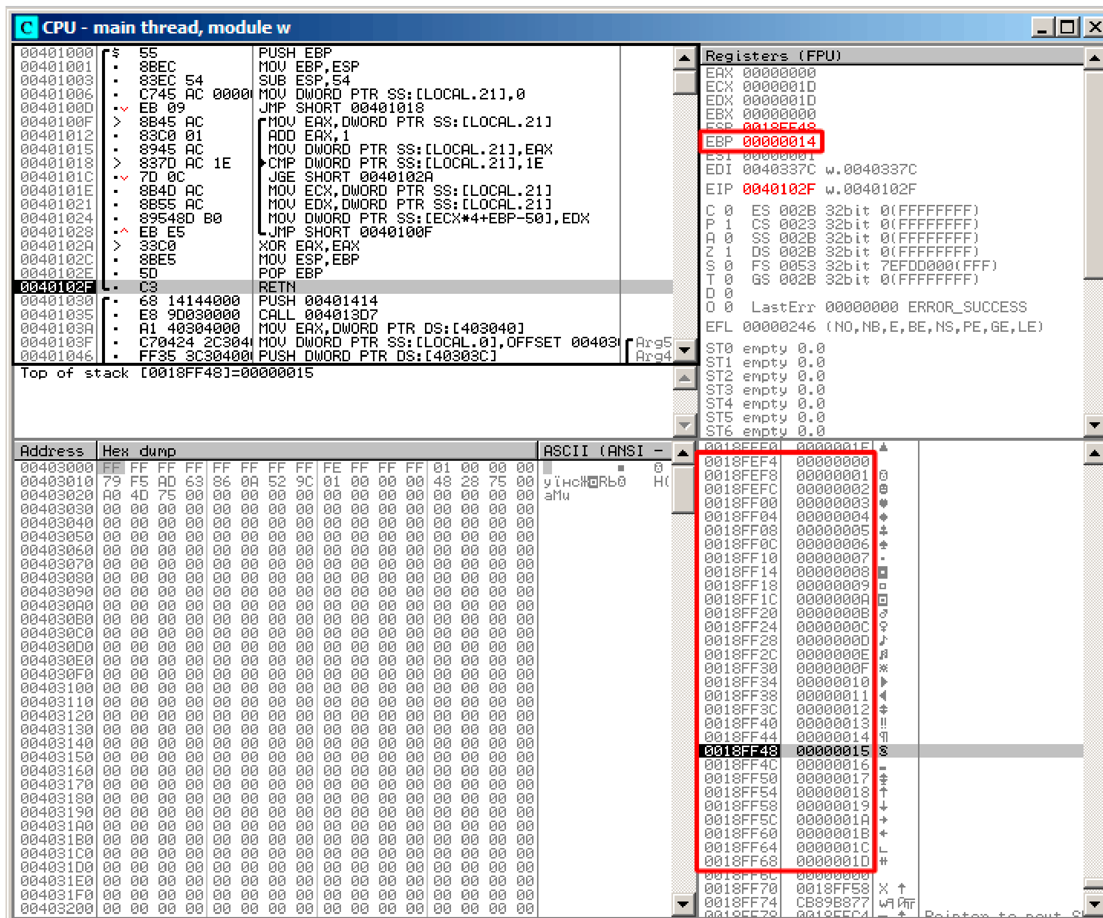


Abbildung 1.90: OllyDbg: nach Wiederherstellung des Wertes von EBP

Nachverfolgen bis zum Ende der Funktion:

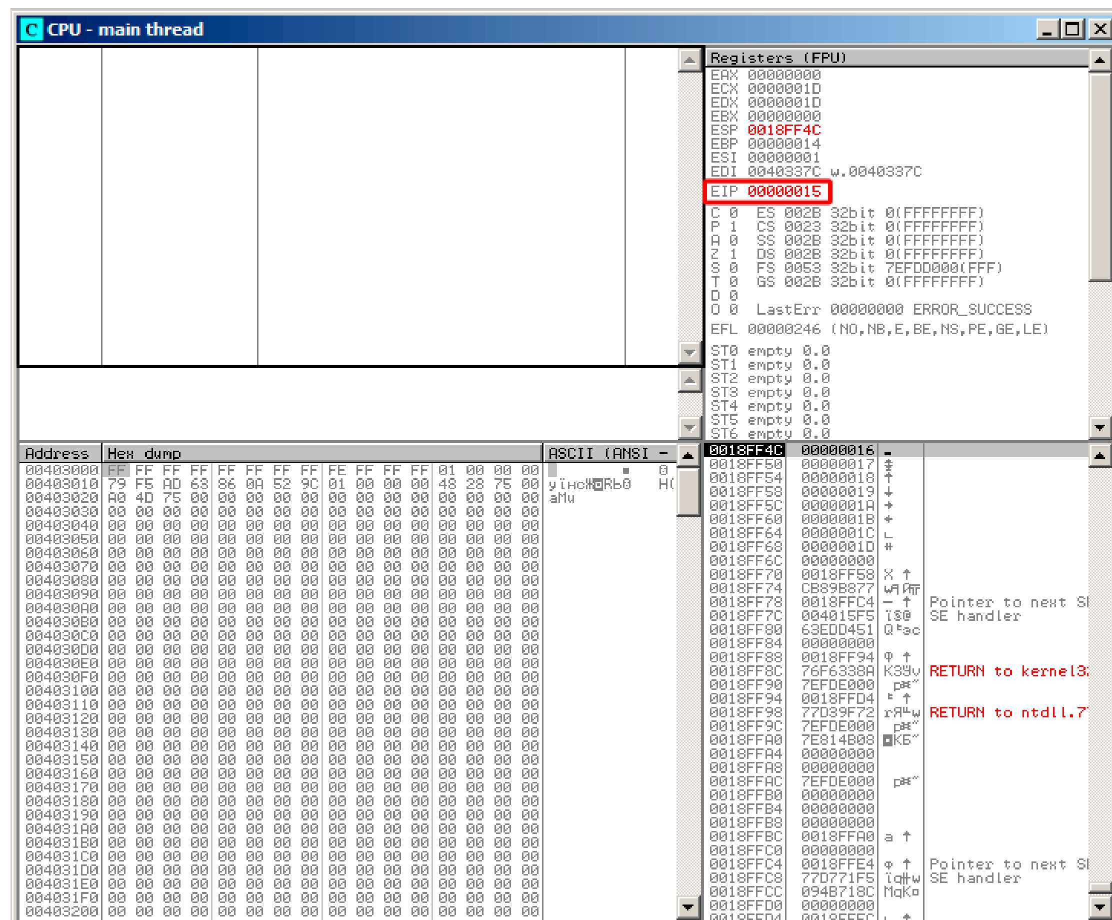


Abbildung 1.91: OllyDbg: EIP wurde wiederhergestellt, aber OllyDbg kann an 0x15 nicht disassemblieren

Richten wir unser Augenmerk auf die Register.

EIP ist jetzt gerade 0x15. Das ist keine gültige Adreses für Code—zumindest nicht für win32 Code! Interessant ist auch, dass das EBP Register 0x14 enthält und ECX sowie EDX jeweils 0x1D

Schauen wir uns das Stacklayout etwas genauer an.

Nachdem der Control Flow an `main()` übergeben worde ist, wurde der Wert in EBP auf dem Stack abgelegt. Danach wurden 84 Byte für das Array und die Variable i reserviert. Das entspricht `(20+1)*sizeof(int)`. ESP zeigt jetzt auf die Variable `_i` im lokalen Stack und nach der Ausführung von `PUSH something` scheint sich `something` neben `_i` zu befinden.

Hier ist das Stacklayout während der Control Flow in der `main()` ist:

ESP	4 Byte reserviert für Variable i
ESP+4	80 Byte reserviert für Array a[20]
ESP+84	sichere Wert von EBP
ESP+88	Rücksprungadresse

Der Befehl a[19]=something schreibt den letzten *int* innerhalb der Grenzen des Arrays (bis hierhin ist alles in Ordnung!). Der Befehl a[20]=something schreibt *something* an die Stelle, an der der EBP gespeichert ist.

Sehen wir uns den Zustand der Register im Moment des Absturzes an. In unserem Fall wurde 20 in das zwanzigste Element geschrieben. Am Ende der Funktion stellt der Funktionsepilog den originalen Wert von EBP wieder her. (20 dezimal entspricht 0x14 hexadezimal). Danach wird RET ausgeführt, was äquivalent zum Befehl POP EIP ist.

Der Befehl RET nimmt die Rücksprungadresse vom Stack (das ist die Adresse in CRT, die main() aufgerufen hat) und speichert hier den Wert 21 (0x15 hexadezimal). Die CPU springt an die Adresse 0x15, aber hier befindet sich kein ausführbarer Code, sodass eine Exception geworfen wird.

Dies nennt man einen *Buffer Overflow*[117].

Ersetzt man das *int* Array durch einen String (*char* Array) und erzeugt absichtlich einen langen String und übergibt ihn im Programm an eine Funktion, die die Länge des Strings nicht prüft und ihn in einen kurzen Buffer kopiert, kann man das Programm zwingen an eine bestimmte Adresse zu springen. In der Realität ist dieses Verhalten nicht so einfach zu erzeugen, funktioniert aber von Prinzip her genau wie hier. Ein klassischer Artikel dazu:[Aleph One, *Smashing The Stack For Fun And Profit*, (1996)][118].

GCC

Kompilieren wir denselben Code mit GCC 4.4.1, erhalten wir:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 60h ; 96
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
```

[117] wikipedia
[118] Auch verfügbar als http://yurichev.com/mirrors/phrack/p49-0x0e.txt

```
loc_80483D1:
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

Lässt man das Programm unter Linux laufen, lautet das Ergebnis: `Segmentation fault`.

Wenn wir es mit dem GDB Debugger laufen lassen, erhalten wir das Folgende:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388       -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0       0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Die Registerwerte unterscheiden sich geringfügig vom win32 Beispiel, da auch das Stacklayout ein wenig anders ist.

1.20.3 Schutz vor Buffer Overflows

Es gibt verschiedene Möglichkeiten um sich vor solchen Problemen zu schützen, unabhängig von der Unachtsamkeit des C/C++ Programmierers. MSVC kennt Optionen wie[119]:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Eine Methode ist eine Zufallszahl zwischen die lokalen Variablen auf dem Stack am Funktionsprolog zu schreiben und diesen im Funktionsepilog vor dem Beenden der

[119] Compilerseitiger Schutz vor Buffer Overflows: wikipedia.org/wiki/Buffer_overflow_protection

Funktion zu überprüfen. Wenn der Wert nicht identisch ist, sollte der letzte RET Befehl nicht ausgeführt werden, sondern das Programm angehalten werden. Der Prozess wird anhalten, aber das ist deutlich besser als eine Fernattacke auf Ihren Rechner.

Die Zufallszahl wird auch „canary“ (dt. Kanarienvogel) genannt. Der Begriff stammt von den Kanarienvögeln der Minenarbeiter[120], die früher benutzt wurde, um giftige Gase schnell zu erkennen

Kanarienvögel reagieren sehr sensibel auf Grubengase und werden bei Gefahr sehr nervös oder sterben sogar.

Wenn wir unser einfaches Arraybeispiel in MSVC[121] mit Optionen RTC1 und RTCs kompilieren (1.20.1 on page 309) finden wir einen Aufruf von `@_RTC_CheckStackVars@8`, eine Funktion am Ende der Funktion, die prüft, ob der „canary“ korrekt ist.

Schauen wir uns an, wie GCC die Sache handhabt. Betrachten wir ein Beispiel mit `alloca()` (1.7.3 on page 47):

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

Ohne zusätzliche Optionen fügt GCC 4.7.3 standardmäßig dem Code einen „canary“ zum Überprüfen hinzu:

Listing 1.208: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
```

[120] wikipedia.org/wiki/Domestic_canary#Miner.27s_canary
[121] Microsoft Visual C++

```
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0  ; "hi! %d, %d, %d\n"
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs:20     ; canary
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20     ; prüfe canary
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5:
        call    __stack_chk_fail
```

Der Zufallswert befindet sich in `gs:20`. Er wird auf den Stack geschrieben und am Ende der Funktion wird der Wert auf dem Stack mit dem korrekten „canary" in `gs:20` verglichen. Wenn die Werte ungleich sind, wird die Funktion `__stack_chk_fail` aufgerufen und wir erkennen in der Konsole in etwa das Folgende (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586    /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586    /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586    /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0          [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602    /lib/i386-linux-gnu/⤦
    ↳ libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602    /lib/i386-linux-gnu/⤦
    ↳ libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602    /lib/i386-linux-gnu/⤦
    ↳ libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781    /lib/i386-linux-gnu/libc⤦
    ↳ -2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781    /lib/i386-linux-gnu/libc⤦
    ↳ -2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781    /lib/i386-linux-gnu/libc⤦
```

```
    ↳ -2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0          [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794    /lib/i386-linux-gnu/ld↙
    ↳ -2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
```

gs ist das sogenannte Segmentregister. Diese Register wurden zu Zeiten von MS-DOS und DOS-Erweiterungen häufig verwendet. Heute ist sein Zweck ein anderer: Kurz gesagt, zeigt das gs Register in Linux stets auf den TLS[122] (6.2 on page 591)–hier werden threadspezifische Informationen gespeichert. In win32 spielt das fs Register übrigens die gleiche Rolle und zeigt stets auf TIB[123][124].

Mehr Informationen finden sich im Quellcode des Linux Kernels (zumindest in der Version 3.11), in
arch/x86/include/asm/stackprotector.h wird diese Variable in den Kommentaren beschrieben.

Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

Betrachten wir nochmals unser einfaches Arraybeispiel(1.20.1 on page 309), erkennen wir nun wie LLVM die Korrektheit des „canary“ überprüft:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
```

[122]Thread Local Storage
[123]Thread Information Block
[124]wikipedia.org/wiki/Win32_Thread_Information_Block

```
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH     {R4-R7,LR}
    ADD      R7, SP, #0xC
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
    MOVW     R0, #aObjc_methtype ; "objc_methtype"
    MOVS     R2, #0
    MOVT.W   R0, #0
    MOVS     R5, #0
    ADD      R0, PC
    LDR.W    R8, [R0]
    LDR.W    R0, [R8]
    STR      R0, [SP,#0x64+canary]
    MOVS     R0, #2
    STR      R2, [SP,#0x64+var_64]
    STR      R0, [SP,#0x64+var_60]
    MOVS     R0, #4
    STR      R0, [SP,#0x64+var_5C]
    MOVS     R0, #6
    STR      R0, [SP,#0x64+var_58]
    MOVS     R0, #8
    STR      R0, [SP,#0x64+var_54]
    MOVS     R0, #0xA
    STR      R0, [SP,#0x64+var_50]
    MOVS     R0, #0xC
    STR      R0, [SP,#0x64+var_4C]
    MOVS     R0, #0xE
    STR      R0, [SP,#0x64+var_48]
    MOVS     R0, #0x10
    STR      R0, [SP,#0x64+var_44]
    MOVS     R0, #0x12
    STR      R0, [SP,#0x64+var_40]
    MOVS     R0, #0x14
    STR      R0, [SP,#0x64+var_3C]
    MOVS     R0, #0x16
    STR      R0, [SP,#0x64+var_38]
    MOVS     R0, #0x18
    STR      R0, [SP,#0x64+var_34]
    MOVS     R0, #0x1A
    STR      R0, [SP,#0x64+var_30]
    MOVS     R0, #0x1C
    STR      R0, [SP,#0x64+var_2C]
    MOVS     R0, #0x1E
    STR      R0, [SP,#0x64+var_28]
    MOVS     R0, #0x20
    STR      R0, [SP,#0x64+var_24]
    MOVS     R0, #0x22
    STR      R0, [SP,#0x64+var_20]
    MOVS     R0, #0x24
```

```
    STR      R0, [SP,#0x64+var_1C]
    MOVS     R0, #0x26
    STR      R0, [SP,#0x64+var_18]
    MOV      R4, 0xFDA ; "a[%d]=%d\n"
    MOV      R0, SP
    ADDS     R6, R0, #4
    ADD      R4, PC
    B        loc_2F1C

; second loop begin

loc_2F14
    ADDS     R0, R5, #1
    LDR.W    R2, [R6,R5,LSL#2]
    MOV      R5, R0

loc_2F1C
    MOV      R0, R4
    MOV      R1, R5
    BLX      _printf
    CMP      R5, #0x13
    BNE      loc_2F14
    LDR.W    R0, [R8]
    LDR      R1, [SP,#0x64+canary]
    CMP      R0, R1
    ITTTT EQ           ; ist canary immernoch korrekt?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      ___stack_chk_fail
```

Zunächst hat LLVM die Schleife entwickelt und alle Werte werden nacheinander vorberechnet in ein Array geschrieben, da LLVM dies für schneller hält. Befehle im ARM mode können helfen, das noch schneller auszuführen und dies herauszufinden könnte Ihre Aufgabe sein.

Am Ende der Funktion sehen wir den Vergleich der beiden „canaries“–dem im lokalen Stack und dem richtigen, auf den R8 zeigt. Wenn sie gleich sind wird durch `ITTTT EQ` ein Block aus vier Befehlen ausgeführt, der 0 nach `R0` schreibt, den Funktionsepilog durchführt und dann beendet. Wenn die „canaries“ ungleich sind, wird der Block übersprungen und es wird zu
`___stack_chk_fail` gesprungen und die Ausführung wird angehalten.

1.20.4 Noch ein Wort zu Arrays

Wir verstehen nun warum es nicht möglich ist etwas wie das Folgende in C/C++ Code zu schreiben:

```
void f(int size)
{
    int a[size];
```

```
...
};
```

Das liegt daran, dass der Compiler die exakte Größe des Arrays zur Compilerzeit kennen muss, um Platz auf dem lokalen Stack zu reservieren.

Wenn man ein Array beliebiger Größe benötigt, muss es über `malloc()` angelegt werden und dann über den reservieren Speicherblock als Arrays von Variablen des benötigten Typs angesprochen werden.

Oder man verwendet das C99 Standardfeature [*ISO/IEC 9899:TC3 (C C99 standard)*, (2007)6.7.5/2], dass intern wie *alloca()* (1.7.3 on page 47) arbeitet.

Es ist auch möglich, C-Bibliotheken zu verwenden, die als Garbagecollector fungieren. Des Weiteren gibt es auch Bibliotheken für C++, die intelligente Pointer unterstützen.

1.20.5 Array von Stringpointern

Hier ist ein Beispiel für ein Array aus Pointern.

Listing 1.209: Get month name

```
#include <stdio.h>

const char* month1[]=
{
        "January", "February", "March", "April",
        "May", "June", "July", "August",
        "September", "October", "November", "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
        return month1[month];
};
```

x64

Listing 1.210: Optimierender MSVC 2013 x64

```
_DATA   SEGMENT
month1  DQ      FLAT:$SG3122
        DQ      FLAT:$SG3123
        DQ      FLAT:$SG3124
        DQ      FLAT:$SG3125
        DQ      FLAT:$SG3126
        DQ      FLAT:$SG3127
        DQ      FLAT:$SG3128
        DQ      FLAT:$SG3129
        DQ      FLAT:$SG3130
        DQ      FLAT:$SG3131
```

```
        DQ      FLAT:$SG3132
        DQ      FLAT:$SG3133
$SG3122 DB      'January', 00H
$SG3123 DB      'February', 00H
$SG3124 DB      'March', 00H
$SG3125 DB      'April', 00H
$SG3126 DB      'May', 00H
$SG3127 DB      'June', 00H
$SG3128 DB      'July', 00H
$SG3129 DB      'August', 00H
$SG3130 DB      'September', 00H
$SG3156 DB      '%s', 0aH, 00H
$SG3131 DB      'October', 00H
$SG3132 DB      'November', 00H
$SG3133 DB      'December', 00H
_DATA   ENDS

month$ = 8
get_month1 PROC
        movsxd  rax, ecx
        lea     rcx, OFFSET FLAT:month1
        mov     rax, QWORD PTR [rcx+rax*8]
        ret     0
get_month1 ENDP
```

Der Code ist sehr einfach:

- Der erste MOVSXD Befehl kopiert einen 32-Bit-Wert aus ECX (wohin das $month$ Argument übergeben wird) nach RAX und erweitert ihn um ein Vorzeichen (da $month$ vom Typ *int* ist).

  Der Grund für die Erweiterung ist, dass dieser 32-Bit-Wert in Berechnungen mit anderen 64-Bit-Werten zusammen verwendet wird. [125].

- Danach wird die Adresse der Pointertabelle nach RCX geladen.

- Schließlich wird der Eingabewert ($month$) mit 9 multipliziert und zur Adresse addiert. Es gilt: wir befinden uns in einer 64-Bit-Umgebung und alle Adressen (oder Pointer) benötigen genau 64 Bit (oder 8 Byte) zum Speichern. Deshalb ist jedes Element der Tabelle 8 Byte breit. Ebenfalls deshalb müssen, um ein spezifisches Element auszuwählen $month \cdot 8$ Bytes vom Start weg übersprungen werden. Genau das tut MOV. Zusätzlich lädt dieser Befehl auch ein Element in diese Adresse. Für 1 würde das Element ein Pointer auf den String „Februar" sein, etc.

Optimierender GCC 4.9 ist noch effizienter: [126]:

Listing 1.211: Optimierender GCC 4.9 x64

[125] Es ist seltsam, aber negative Arrayindizes für $month$ können hier verwendet werden (negative Arrayindizes werden später erklärt: **??** on page ??). Wenn dies passiert, wird der negative Eingabewert vom Typ *int* korrekt um ein Vorzeichen erweitert und das zugehörige Element vor der Tabelle wird ausgewählt. Ohne Vorzeichenerweiterung würde es nicht korrekt funktionieren.

[126] „0+" blieb im Listing,da der GCC Assembler-Output nicht sauber genug ist, um es zu eliminieren.

```
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
```

32-bit MSVC

Kompilieren wir den Code mit dem 32-Bit-MSVC-Compiler:

Listing 1.212: Optimierender MSVC 2013 x86

```
_month$ = 8
_get_month1 PROC
        mov     eax, DWORD PTR _month$[esp-4]
        mov     eax, DWORD PTR _month1[eax*4]
        ret     0
_get_month1 ENDP
```

Der Eingabewert darf nicht zu einem 64-Bit-Wert erweitert werden und wird so wie er ist verwendet.

Er wird mit 4 multipliziert, da die Tabellenelemente 32 Bit (oder 4 Byte) breit sind.

32-bit ARM

ARM im ARM mode

Listing 1.213: Optimierender Keil 6/2013 (ARM Modus)

```
get_month1 PROC
        LDR      r1,|L0.100|
        LDR      r0,[r1,r0,LSL #2]
        BX       lr
        ENDP

|L0.100|
        DCD      ||.data||

        DCB      "January",0
        DCB      "February",0
        DCB      "March",0
        DCB      "April",0
        DCB      "May",0
        DCB      "June",0
        DCB      "July",0
        DCB      "August",0
        DCB      "September",0
        DCB      "October",0
        DCB      "November",0
        DCB      "December",0

        AREA ||.data||, DATA, ALIGN=2
month1
        DCD      ||.conststring||
```

```
        DCD      ||.conststring||+0x8
        DCD      ||.conststring||+0x11
        DCD      ||.conststring||+0x17
        DCD      ||.conststring||+0x1d
        DCD      ||.conststring||+0x21
        DCD      ||.conststring||+0x26
        DCD      ||.conststring||+0x2b
        DCD      ||.conststring||+0x32
        DCD      ||.conststring||+0x3c
        DCD      ||.conststring||+0x44
        DCD      ||.conststring||+0x4d
```

Die Adresse der Tabelle wird nach R1 geladen.

Der ganze Rest wird mit lediglich einem LDR Befehl erledigt.

Danach wird der Eingabewert $month$ um zwei Bit nach links verschoben (dies entspricht einer Multiplikation mit 4), schließlich zu R1 addiert (wo sich die Adresse der Tabelle befindet) und schlußendlich wird ein Tabellenelement aus dieser Adresse geladen.

Das 32-Bit-Tabellenelement wird aus der Tabelle nach R0 geladen.

ARM im Thumb mode

Der Code ist fast identisch, aber weniger dicht, denn der LSL Suffix kann hier nicht an den LDR Befehl angehängt werden:

```
get_month1 PROC
        LSLS     r0,r0,#2
        LDR      r1,|L0.64|
        LDR      r0,[r1,r0]
        BX       lr
        ENDP
```

ARM64

Listing 1.214: Optimierender GCC 4.9 ARM64

```
get_month1:
        adrp    x1, .LANCHOR0
        add     x1, x1, :lo12:.LANCHOR0
        ldr     x0, [x1,w0,sxtw 3]
        ret

.LANCHOR0 = . + 0
        .type   month1, %object
        .size   month1, 96
month1:
        .xword  .LC2
        .xword  .LC3
        .xword  .LC4
        .xword  .LC5
```

```
        .xword  .LC6
        .xword  .LC7
        .xword  .LC8
        .xword  .LC9
        .xword  .LC10
        .xword  .LC11
        .xword  .LC12
        .xword  .LC13
.LC2:
        .string "January"
.LC3:
        .string "February"
.LC4:
        .string "March"
.LC5:
        .string "April"
.LC6:
        .string "May"
.LC7:
        .string "June"
.LC8:
        .string "July"
.LC9:
        .string "August"
.LC10:
        .string "September"
.LC11:
        .string "October"
.LC12:
        .string "November"
.LC13:
        .string "December"
```

Die Adresse der Tabelle wird mit ADRP/ADD nach X1 geladen.

Dann wird das zugehörige Element mit einem LDR ausgewählt, das W0 nimmt (das Register, in dem sich der Eingabewert $month$ befindet), es um 3 Bit nach links verschiebt (was einer Multiplikation mit 8 entspricht), um ein Vorzeichen erweitert (das bedeutet der Suffix „sxtw“) und dann zu X0 addiert. Schließlich wird der 64-Bit-Wert aus der Tabelle nach X0 geladen.

MIPS

Listing 1.215: Optimierender GCC 4.4.5 (IDA)

```
get_month1:
; lade Adresse der Tabelle nach $v0:
                la      $v0, month1
; nimm den Eingabewert und multipliziere mit 4:
                sll     $a0, 2
; addiere Adresse der Tabelle und berechneten Wert:
                addu    $a0, $v0
; lade Tabellenelement an dieser Adresse nach $v0:
```

```
                lw      $v0, 0($a0)
; Rückgabe
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP

                .data # .data.rel.local
                .globl month1
month1:         .word aJanuary          # "January"
                .word aFebruary         # "February"
                .word aMarch            # "March"
                .word aApril            # "April"
                .word aMay              # "May"
                .word aJune             # "June"
                .word aJuly             # "July"
                .word aAugust           # "August"
                .word aSeptember        # "September"
                .word aOctober          # "October"
                .word aNovember         # "November"
                .word aDecember         # "December"

                .data # .rodata.str1.4
aJanuary:       .ascii "January"<0>
aFebruary:      .ascii "February"<0>
aMarch:         .ascii "March"<0>
aApril:         .ascii "April"<0>
aMay:           .ascii "May"<0>
aJune:          .ascii "June"<0>
aJuly:          .ascii "July"<0>
aAugust:        .ascii "August"<0>
aSeptember:     .ascii "September"<0>
aOctober:       .ascii "October"<0>
aNovember:      .ascii "November"<0>
aDecember:      .ascii "December"<0>
```

Array Overflow

Unsere Funktion akzeptiert Werte im Bereich von 0 bis 11, aber was, wenn 12 übergeben wird? Es gibt an dieser Stelle in der Tabelle kein Element.

Die Funktion wird also irgendeinen dort befindlichen Wert laden und ihn zurückgeben.

Kurz danach kann eine andere Funktion versuchen, einen Textstring von dieser Adresse zu laden und könnte abstürzen.

Kompilieren wir das Beipsiel mit MSVC für win64 und öffnen es in IDA, um zu sehen was der Linker hinter der Tabelle angelegt hat:

Listing 1.216: Executable file in IDA

```
off_140011000   dq offset aJanuary_1    ; DATA XREF: .text:0000000140001003
                                        ; "January"
                dq offset aFebruary_1   ; "February"
                dq offset aMarch_1      ; "March"
```

```
                 dq offset aApril_1      ; "April"
                 dq offset aMay_1        ; "May"
                 dq offset aJune_1       ; "June"
                 dq offset aJuly_1       ; "July"
                 dq offset aAugust_1     ; "August"
                 dq offset aSeptember_1  ; "September"
                 dq offset aOctober_1    ; "October"
                 dq offset aNovember_1   ; "November"
                 dq offset aDecember_1   ; "December"
aJanuary_1       db 'January',0          ; DATA XREF: sub_140001020+4
                                         ; .data:off_140011000
aFebruary_1      db 'February',0         ; DATA XREF: .data:0000000140011008
                 align 4
aMarch_1         db 'March',0            ; DATA XREF: .data:0000000140011010
                 align 4
aApril_1         db 'April',0            ; DATA XREF: .data:0000000140011018
```

Die Monatsnamen befinden sich direkt dahinter.

Unser Programm ist winzig, sodass hier nicht viele Daten im Datensegment abgelegt werden müssen, nur die Monatsnamen, Wir stellen aber fest, dass sich hier irgendetwas befinden könnte, was der Linker hier zufällig platziert hat.

Was also, falls 12 an die Funktion übergeben wird? Das 13. Element wird zurückgegeben.

Schauen wir uns an, wie die CPU die Bytes dort wie einen 64-Bit-Wert behandelt:

Listing 1.217: Executable file in IDA

```
off_140011000    dq offset qword_140011060
                                         ; DATA XREF: .text:0000000140001003
                 dq offset aFebruary_1   ; "February"
                 dq offset aMarch_1      ; "March"
                 dq offset aApril_1      ; "April"
                 dq offset aMay_1        ; "May"
                 dq offset aJune_1       ; "June"
                 dq offset aJuly_1       ; "July"
                 dq offset aAugust_1     ; "August"
                 dq offset aSeptember_1  ; "September"
                 dq offset aOctober_1    ; "October"
                 dq offset aNovember_1   ; "November"
                 dq offset aDecember_1   ; "December"
qword_140011060 dq 797261756E614Ah       ; DATA XREF: sub_140001020+4
                                         ; .data:off_140011000
aFebruary_1      db 'February',0         ; DATA XREF: .data:0000000140011008
                 align 4
aMarch_1         db 'March',0            ; DATA XREF: .data:0000000140011010
```

Dieser Wert ist 0x797261756E614A.

Kurz danach könnte eine andere Funktion (möglicherweise eine, die Strings verarbeitet) versuchen, Bytes von dieser Adresse zu lesen, weil sie hier einen C-String erwartet.

Höchstwahrscheinlich wird dies zu einem Absturz führen, da der Wert keine gültige Adresse sein wird.

Schutz vor Array Overflows

> Alles, was schiefgehen kann, wird auch schiefgehen
>
> Murphy's Law

Es ist ein wenig naiv zu erwarten, dass jeder Programmierer, der diese Funktion oder Bibliotek verwendet, nie ein größeres Argument als 11 übergeben wird.

Es gibt einen Ansatz, der besagt „scheitere früh und laut“ oder „scheitere schnell“, und dessen Aussage es ist, dass Probleme so früh wie möglich gemeldet werden und das Programm angehalten werden sollte.

Eine solche Methode in C/C++ sind Assertions.

Wir modifizieren unser Programm, sodass es einen Fehler liefert, wenn ein falscher Wert übergeben wird:

Listing 1.218: assert() added

```
const char* get_month1_checked (int month)
{
        assert (month<12);
        return month1[month];
};
```

Das Assertionmakro prüft zu Beginn der Funktion auf valide Werte und liefert einen Fehler, wenn der Ausdruck falsch ist.

Listing 1.219: Optimierender MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
        DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
        DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
        push    rbx
        sub     rsp, 32
        movsxd  rbx, ecx
        cmp     ebx, 12
        jl      SHORT $LN3@get_month1
        lea     rdx, OFFSET FLAT:$SG3143
        lea     rcx, OFFSET FLAT:$SG3144
        mov     r8d, 29
        call    _wassert
$LN3@get_month1:
        lea     rcx, OFFSET FLAT:month1
```

```
        mov     rax, QWORD PTR [rcx+rbx*8]
        add     rsp, 32
        pop     rbx
        ret     0
get_month1_checked ENDP
```

Tatsächlich ist assert() keine Funktion, sondern ein Makro. Es prüft auf eine Bedingung und übergibt dann auch die Zeilennummer und den Dateinamen an eine andere Funktion, die diese Informationen an den Benutzer weiterleitet.

Wir erkennen hier, dass sowohl Dateiname als auch Bedingung in UTF-16 kodiert sind. Die Zeilennummer wird ebenfalls übergeben (hier: 29).

Dieser Mechanismus ist möglicherweise in allen Compilern der gleiche. GCC erzeugt den folgenden Code:

Listing 1.220: Optimierender GCC 4.9 x64

```
.LC1:
        .string "month.c"
.LC2:
        .string "month<12"

get_month1_checked:
        cmp     edi, 11
        jg      .L6
        movsx   rdi, edi
        mov     rax, QWORD PTR month1[0+rdi*8]
        ret
.L6:
        push    rax
        mov     ecx, OFFSET FLAT:__PRETTY_FUNCTION__.2423
        mov     edx, 29
        mov     esi, OFFSET FLAT:.LC1
        mov     edi, OFFSET FLAT:.LC2
        call    __assert_fail

__PRETTY_FUNCTION__.2423:
        .string "get_month1_checked"
```

Das Makro übergibt praktischerweise in GCC auch den Funktionsnamen.

Es gibt aber nichts umsonst und das gilt auch für solche Überprüfungen.

Sie machen das Programm langsamer, vor allem, wenn das assert() Makro in kleinen zeitkritischen Funktionen verwendet wird.

MSVC zum Beispiel verwendet die Checks in den Debug Builds, aber lässt sie in den Release Builds weg.

Microsoft Windows NT kernels gibt es als „geprüfte“ und „freie“ Builds [127]. Die erste Variante enthält Validierungsprüfungen (daher „geprüft“), die zweite nicht (daher „frei“ von Überprüfungen).

[127] msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx

Natürlich arbeitet der „geprüfte“ Kernel wegen der vielen Überprüfungen langsamer, sodass es normalerweise nur zum Debuggen verwendet wird.

1.20.6 Multidimensionale Arrays

Intern ist ein multidimensionales Array im Prinzip das gleiche wie ein lineares Array.

Da der Speicher eines Rechners linear ist, ist es ein eindimensionales Array. Zur Vereinfachung kann dieses multidimensionales Array leicht als eindimensional dargestellt werden.

Beispielsweise werden die Elemente eines 3x4 Arrays folgendermaßen in einem eindimensionalen Array aus 12 Zellen gespeichert:

Offset im Speicher	Arrayelement
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

Tabelle 1.3: Zweidimensionales Array in eindimensionaler Speicherdarstellung

Auf diese Weise wird jede Zellen des 3*4 Arrays im Speicher abgelegt:

0	1	2	3
4	5	6	7
8	9	10	11

Tabelle 1.4: Speicheradressen jeder Zelle des zweidimensionalen Arrays

Um also die Adresse des benötigten Elements zu berechnen, multiplizieren wir zunächst den ersten Index mit 4 (der Arraybreite) und addieren dann den zweiten Index. Dies nennt man *Zeilenordnung* (engl. row-major order) und diese Methode zur Darstellung von Arrays und Matrizen wird mindestens von C/C++ und Python verwendet. Der Ausdruck row-major order bedeutet: „schreibe zuerst die Elemente der ersten Zeilen, dann die zweite Zeile…und schließlich die Elemente der letzten Zeile“.

Eine andere Methode zur Darstellung heißt *Spaltenordnung* (engl. column-major order) (die Indizes des Arrays werden in umgekehrter Reihenfolge verwendet) und wird zumindest in Fortran, MATLAB und R verwendet. Der Ausdruck column-major

oder bedeutet: „schreibe zuerst die Elemente der ersten Spalte, dann die zweite Spalte…und schließlich die Elemente der letzten Spalte“.

Welche Method ist besser?

Generel ist hinsichtlich Performance und Cachespeicher die beste Methode der Datenorganisation diejenige, in der auf die Elemente sequentiell zugegriffen wird.

Wenn eine Funktion auf Daten zeilenweise zugreift, ist Zeilenordnung besser und umgekehrt.

Beispiel für zweidimensionales Array

Wir werden mit einem Array vom Typ *char* arbeiten, was bedeutet, dass jedes Element nur ein Byte Speicherplatz benötigt.

Beispiel: Zeile füllen

Füllen wir die zweite Zeilen mit den Werten 0..3:

Listing 1.221: Row filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // Array leeren
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // zweite Spalte mit 0..3 füllen:
        for (y=0; y<4; y++)
                a[1][y]=y;
};
```

Alle drei Zeilen sind rot markiert. Wir erkennen, dass die zweite Zeilen nun die Werte 0,1,2 und 3 enthält:

```
Address   Hex dump
00C33370  00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380  02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Abbildung 1.92: OllyDbg: Array ist befüllt

Beipsiel: Spalte füllen

Füllen wir die dritte Spalte mit den Werten 0..2:

Listing 1.222: Column filling example

```
#include <stdio.h>

char a[3][4];

int main()
{
        int x, y;

        // leere Array
        for (x=0; x<3; x++)
                for (y=0; y<4; y++)
                        a[x][y]=0;

        // fülle dritte Spalte mit 0..2:
        for (x=0; x<3; x++)
                a[x][2]=x;
};
```

Die drei Spalten sind hier ebenfalls rot markiert.

Wir erkennen, dass sich in jeder Zeile an der dritten Stelle die Werte 0,1 und 2 befinden.

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E AA EF 31 1E AA EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Abbildung 1.93: OllyDbg: Array ist befüllt

Eindimensionaler Zugriff auf zweidimensionales Array

Wir können uns leicht davon überzeugen, dass es auf mindestens zwei Arten möglich ist, auf ein zweidimensionales Array eindimensional zuzugreifen:

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
{
        return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
        // behandle Eingabearray eindimensional
        // 4 entspricht der Arraybreite
        return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
```

```
{
        // behandle Eingabearray als Pointer,
        // berechne Adresse, lade Wert an dieser Stelle
        // 4 entspricht der Arraybreite
        return *(array+a*4+b);
};

int main()
{
        a[2][3]=123;
        printf ("%d\n", get_by_coordinates1(a, 2, 3));
        printf ("%d\n", get_by_coordinates2(a, 2, 3));
        printf ("%d\n", get_by_coordinates3(a, 2, 3));
};
```

Kompilieren und ausführen: es zeigt korrekte Werte an Was MSVC 2013 getan hat ist faszinierend: alle drei Routinen sind identisch!

Listing 1.223: Optimierender MSVC 2013 x64

```
array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=Adresse des Arrays
; RDX=a
; R8=b
        movsxd  rax, r8d
; EAX=b
        movsxd  r9, edx
; R9=a
        add     rax, rcx
; RAX=b+Adresse des Arrays
        movzx   eax, BYTE PTR [rax+r9*4]
; AL=lade Byte an der Adresse RAX+R9*4=b+Adresse des Arrays+a*4=Adresse des
    Arrays+a*4+b
        ret     0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
```

```
        movsxd  rax, r8d
        movsxd  r9, edx
        add     rax, rcx
        movzx   eax, BYTE PTR [rax+r9*4]
        ret     0
get_by_coordinates1 ENDP
```

GCC erzeugt ebenfalls äquivalente Routinen, aber ein wenig anders:

Listing 1.224: Optimierender GCC 4.9 x64

```
; RDI=Adresse des Arrays
; RSI=a
; RDX=b

get_by_coordinates1:
; erweitere 32-Bit int Eingabewerte "a" und "b" zu 64-Bit-Werten.
        movsx   rsi, esi
        movsx   rdx, edx
        lea     rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=Adresse des Arrays+a*4
        movzx   eax, BYTE PTR [rax+rdx]
; AL=lade Byte an der Adresse RAX+RDX=Adresse des Arrays+a*4+b
        ret

get_by_coordinates2:
        lea     eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
        cdqe
        movzx   eax, BYTE PTR [rdi+rax]
; AL=lade Byte an der Adresse RDI+RAX=Adresse des Arrays+b+a*4
        ret

get_by_coordinates3:
        sal     esi, 2
; ESI=a<<2=a*4
; erweitere 32-Bit int Eingabewerte "a*4" und "b" zu 64-Bit-Werten.
        movsx   rdx, edx
        movsx   rsi, esi
        add     rdi, rsi
; RDI=RDI+RSI=Adresse des Arrays+a*4
        movzx   eax, BYTE PTR [rdi+rdx]
; AL=lade Byte an der Adresse RDI+RDX=Adresse des Arrays+a*4+b
        ret
```

Beispiel: dreidimensionales Array

Mit multidimensionalen Arrays ist es das gleiche.

Wir werden nun mit einem Array vom Typ *int* arbeiten: jedes Element benötigt 4 Byte Speicherplatz.

Sehen wir es uns an:

Listing 1.225: simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

x86

Wir erhalten das Folgende (MSVC 2010):

Listing 1.226: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                ; size = 4
_y$ = 12               ; size = 4
_z$ = 16               ; size = 4
_value$ = 20           ; size = 4
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                   ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                    ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]  ; edx=a + 600*4*x + 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx  ; *(edx+z*4)=Wert
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Nichts Außergewöhnliches. Zur Berechnung des Index' werden in der Formel $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ drei Eingabewerte verwendet, um das multidimensionale Array zu repräsentieren. Vergessen wir nicht, dass der *int* Typ 32 Bit (4 Byte) breit ist, sodass alle Koeffizienten mit 4 multipliziert werden müssen.

Listing 1.227: GCC 4.4.1

```
         public insert
insert   proc near

x        = dword ptr  8
```

```
y        = dword ptr  0Ch
z        = dword ptr  10h
value    = dword ptr  14h

         push    ebp
         mov     ebp, esp
         push    ebx
         mov     ebx, [ebp+x]
         mov     eax, [ebp+y]
         mov     ecx, [ebp+z]
         lea     edx, [eax+eax]    ; edx=y*2
         mov     eax, edx          ; eax=y*2
         shl     eax, 4            ; eax=(y*2)<<4 = y*2*16 = y*32
         sub     eax, edx          ; eax=y*32 - y*2=y*30
         imul    edx, ebx, 600     ; edx=x*600
         add     eax, edx          ; eax=eax+edx=y*30 + x*600
         lea     edx, [eax+ecx]    ; edx=y*30 + x*600 + z
         mov     eax, [ebp+value]
         mov     dword ptr ds:a[edx*4], eax  ; *(a+edx*4)=value
         pop     ebx
         pop     ebp
         retn
insert   endp
```

Der GCC Compiler arbeitet anders.

Für eine der Operationen in der Berechnung ($30y$) produziet GCC Code ohne Multiplikationsbefehle. Das funktioniert wie folgt: $(y + y) \ll 4 - (y + y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$.

So werden für die $30y$ Berechnung nur ein Addierbefehl, eine bitweiser Verschiebebefehl und ein Subtraktionsbefehl verwendet. So geht es schneller.

ARM + Nicht optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

Listing 1.228: Nicht optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

```
_insert

value   = -0x10
z       = -0xC
y       = -8
x       = -4

; reserviere auf dem lokalen Stack Platz für 4 Werte vom Typ int
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9] ; lade Pointer auf Array
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
```

```
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1]   ; R1 - Adresse des Arrayelements
; Block im lokalen Stack freigeben, reserviere Platz für 4 Werte vom Typ int:
ADD     SP, SP, #0x10
BX      LR
```

Nicht optimierender LLVM speichert alle Variablen auf dem lokalen Stack, was redundant ist.

Die Adresse des Arrayelements wird über die eben gezeigte Formel berechnet.

ARM + Optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

Listing 1.229: Optimierender Xcode 4.6.3 (LLVM) (Thumb Modus)

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1,LSL#4  ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC
LDR.W   R9, [R9]          ; R9 = Pointer auf ein Array
MLA.W   R0, R0, R12, R9   ; R0 - x, R12 - 2400, R9 - Pointer auf a. R0=x*2400
    + Pointer auf a
ADD.W   R0, R0, R1,LSL#3  ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + Pointer auf a
    + y*15*8 =
                          ; Pointer auf a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 - z, R3 - Werte. Adresse=R0+z*4 =
                          ; Pointer auf a + y*30*4 + x*600*4 + z*4
BX      LR
```

Die Tricks für das Ersetzen der Multiplikation durch Verschieben, Addieren und Subtrahieren, die wir bereits kennengelernt haben, kommen hier auch vor.

Hier finden wir auch einen für uns neuen Befehl: RSB (*Reverse Subtract*).

Er arbeitet genau wie SUB, aber vertauscht die Operanden vor der Ausführung. Warum?

SUB und RSB sind Befehle, bei denen auf den zweiten Operanden eine bitweise Verschiebung angewendet werden kann: (LSL#4). Dieser Koeffizient kann aber nur auf den zweiten Operanden angewendet werden.

Das ist günstig für kommutative Operationen wie Addition und Multiplikation (die Operanden können vertauscht werden, ohne das Ergebnis zu verändern).

Subtraktion dagegen ist nicht kommutativ, weshalb für diese Fälle RSB existiert.

MIPS

Das Beispiel ist sehr klein, sodass der GCC Compiler entschieden hat das Array a im 64KiB Platz abzulegen, um es durch den globalen Pointer zugreifbar zu machen.

Listing 1.230: Optimierender GCC 4.4.5 (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=Wert
                sll     $v0, $a0, 5
; $v0 = $a0<<5 = x*32
                sll     $a0, 3
; $a0 = $a0<<3 = x*8
                addu    $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
                sll     $v1, $a1, 5
; $v1 = $a1<<5 = y*32
                sll     $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
                sll     $a1, 1
; $a1 = $a1<<1 = y*2
                subu    $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
                subu    $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
                la      $gp, __gnu_local_gp
                addu    $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
                addu    $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; lade Adresse der Tabelle:
                lw      $v0, (a & 0xFFFF)($gp)
; multipliziere Index mit 4 , um das Arrayelement zu suchen:
                sll     $a0, 2
; addiere multiplizierten Index und Tabellenadresse:
                addu    $a0, $v0, $a0
; speichere Wert in Tabelle und beende:
                jr      $ra
                sw      $a3, 0($a0)

                .comm a:0x1770
```

Weitere Beispiele

Der Bildschirm wird als 2D-Array dargestellt, aber der Videopuffer ist ein lineares 1D-Array. Wir betrachten hier näher: **??** on page ??.

Ein anderes Beispiel in diesem Buch ist das Spiel Minesweeper: das Feld ist auch ein zweidimensionales Array: **??** on page ??.

1.20.7 Strings als zweidimensionales Array

Betrachten wir erneut die Funktion, die den Namen eines Monats zurückgibt: Listing.1.209.

Wie man sieht wird mindestens eine Befehl benötigt, der einen Wert aus dem Speicher lädt, um den Pointer auf den String, der den Monatsnamen enthält, vorzubereiten.

Ist es möglich diesen Speicherzugriff loszuwerden?

Die Antwort ist: ja, wenn man die Liste aus String als zweidimensionales Array darstellt:

```
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
        { 'J','a','n','u','a','r','y',  0,  0,  0 },
        { 'F','e','b','r','u','a','r','y',  0,  0 },
        { 'M','a','r','c','h',  0,  0,  0,  0,  0 },
        { 'A','p','r','i','l',  0,  0,  0,  0,  0 },
        { 'M','a','y',  0,  0,  0,  0,  0,  0,  0 },
        { 'J','u','n','e',  0,  0,  0,  0,  0,  0 },
        { 'J','u','l','y',  0,  0,  0,  0,  0,  0 },
        { 'A','u','g','u','s','t',  0,  0,  0,  0 },
        { 'S','e','p','t','e','m','b','e','r',  0 },
        { 'O','c','t','o','b','e','r',  0,  0,  0 },
        { 'N','o','v','e','m','b','e','r',  0,  0 },
        { 'D','e','c','e','m','b','e','r',  0,  0 }
};

// in 0..11 range
const char* get_month2 (int month)
{
        return &month2[month][0];
};
```

Hier ist was wir erhalten:

Listing 1.231: Optimierender MSVC 2013 x64

```
month2  DB      04aH
        DB      061H
        DB      06eH
        DB      075H
```

```
        DB      061H
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
...

get_month2 PROC
; erweitere Eingabewert um Vorzeichen und wandle um in 64-Bit-Wert
        movsxd  rax, ecx
        lea     rcx, QWORD PTR [rax+rax*4]
; RCX=Monat+Monat*4=Monat*5
        lea     rax, OFFSET FLAT:month2
; RAX=Pointer auf die Tabelle
        lea     rax, QWORD PTR [rax+rcx*2]
; RAX=Pointer auf die Tabelle + RCX*2=Pointer auf die Tabelle +
    Monat*5*2=Pointer auf die Tabelle + Monat*10
        ret     0
get_month2 ENDP
```

Es gibt überhaupt keine Speicherzugriffe. Alles, was diese Funktion tut, ist einen Pointer zu berechnen, der auf den ersten Buchstaben des Monats zeigt:$pointer_to_the_table + month \cdot 10$.

Es gibt auch zwei LEA Befehle, die wie mehrere MUL und MOV Befehle funktionieren.

Die Breite des Arrays beträgt 10 Byte.

Der längste String im Beispiel—„September"—hat eine Länge von 9 Byte zuzüglich einer terminierenden Null, also insgesamt 10 Byte.

Die übrigen Monatsnamen werden mit Zerobytes aufgefüllt, sodass alle denselben Speicherplatz (10 Byte) benötigen.

Dadurch arbeitet unsere Funktion noch schneller, denn die Startadresse jedes Strings kann so einfach berechnet werden.

Optimierender GCC 4.9 kann sogar noch kürzeren Code erzeugen:

Listing 1.232: Optimierender GCC 4.9 x64

```
        movsx   rdi, edi
        lea     rax, [rdi+rdi*4]
        lea     rax, month2[rax+rax]
        ret
```

LEA wird hier auch für die Multiplikation mit 10 verwendet.

Nicht optimierende Compiler führen die Multiplikation anders durch.

Listing 1.233: Nicht optimierender GCC 4.9 x64

```
get_month2:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
```

```
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
; RDX = erweitere Eingabewert um Vorzeichen
        mov     rax, rdx
; RAX = Monat
        sal     rax, 2
; RAX = Monat<<2 = Monat*4
        add     rax, rdx
; RAX = RAX+RDX = Monat*4+Monat = Monat*5
        add     rax, rax
; RAX = RAX*2 = Monat*5*2 = Monat*10
        add     rax, OFFSET FLAT:month2
; RAX = Monat*10 + Pointer auf die Tabelle
        pop     rbp
        ret
```

Nicht optimierender MSVC verwendet nur den IMUL Befehl:

Listing 1.234: Nicht optimierender MSVC 2013 x64

```
month$ = 8
get_month2 PROC
        mov     DWORD PTR [rsp+8], ecx
        movsxd  rax, DWORD PTR month$[rsp]
; RAX = Eingabewert um Vorzeichen und auf 64 Bit erweitern
        imul    rax, rax, 10
; RAX = RAX*10
        lea     rcx, OFFSET FLAT:month2
; RCX = Pointer auf die Tabelle
        add     rcx, rax
; RCX = RCX+RAX = Pointer auf die Tabelle+Monat*10
        mov     rax, rcx
; RAX = Pointer auf die Tabelle+Monat*10
        mov     ecx, 1
; RCX = 1
        imul    rcx, rcx, 0
; RCX = 1*0 = 0
        add     rax, rcx
; RAX = Pointer auf die Tabelle+Monat*10 + 0 = Pointer auf die
   Tabelle+Monat*10
        ret     0
get_month2 ENDP
```

Eine Sache hier ist seltsam: warum wird die Multiplikation mit null und die Addition von null zum Endergebnis hinzugefügt?

Dies sieht wie ein Fehler im Codegenerator des Compilers aus, der nicht durch die Tests des Compilers abgefangen wurde. (Trotzdem funktioniert der erzeugte Code korrekt.)

Wir betrachten solche Codes ganz bewußt, damit der Leser sich klarmacht, dass man sich über solche Merkwürdigkeiten und Artefakte des Compilers nicht allzu sehr wundern soll.

32-bit ARM

Optimierender Keil im Thumb mode verwendet zur Multiplikation den Befehl MULS:

Listing 1.235: Optimierender Keil 6/2013 (Thumb Modus)

```
; R0 = Monat
        MOVS     r1,#0xa
; R1 = 10
        MULS     r0,r1,r0
; R0 = R1*R0 = 10*Monat
        LDR      r1,|L0.68|
; R1 = Pointer auf die Tabelle
        ADDS     r0,r0,r1
; R0 = R0+R1 = 10*Monat + Pointer auf die Tabelle
        BX       lr
```

Optimierender Keil für ARM mode verwendet Additions- und Schiebebefehle:

Listing 1.236: Optimierender Keil 6/2013 (ARM Modus)

```
; R0 = Monat
        LDR      r1,|L0.104|
; R1 = Pointer auf die Tabelle
        ADD      r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = Monat*5
        ADD      r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = Pointer auf die Tabelle + Monat*5*2 = Pointer auf die
    Tabelle + Monat*10
        BX       lr
```

ARM64

Listing 1.237: Optimierender GCC 4.9 ARM64

```
; W0 = Monat
        sxtw    x0, w0
; X0 = vorzeichenerweiterter Eingabewert
        adrp    x1, .LANCHOR1
        add     x1, x1, :lo12:.LANCHOR1
; X1 = Pointer auf die Tabelle
        add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
        add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = Pointer auf die Tabelle + X0*10
        ret
```

SXTW wird für Vorzeichenerweiterung und Übertragung von 32-Bit-Werten in 64-Bit-Werte und das Speichern in X0 verwendet.

Das ADRP/ADD Paar wird für das Laden der Adresse der Tabelle verwendet.

Der ADD Befehl trägt auch den LSL Suffix, der bei der Multiplikation hilft.

MIPS

Listing 1.238: Optimierender GCC 4.4.5 (IDA)

```
                .globl get_month2
get_month2:
; $a0=Monat
                sll     $v0, $a0, 3
; $v0 = $a0<<3 = Monat*8
                sll     $a0, 1
; $a0 = $a0<<1 = Monat*2
                addu    $a0, $v0
; $a0 = Monat*2+Monat*8 = Monat*10
; lade Adresse der Tabelle:
                la      $v0, month2
; summiere Tabellenadressen und berechneten Index:
                jr      $ra
                addu    $v0, $a0

month2:         .ascii "January"<0>
                .byte 0, 0
aFebruary:      .ascii "February"<0>
                .byte    0
aMarch:         .ascii "March"<0>
                .byte 0, 0, 0, 0
aApril:         .ascii "April"<0>
                .byte 0, 0, 0, 0
aMay:           .ascii "May"<0>
                .byte 0, 0, 0, 0, 0, 0
aJune:          .ascii "June"<0>
                .byte 0, 0, 0, 0, 0
aJuly:          .ascii "July"<0>
                .byte 0, 0, 0, 0, 0
aAugust:        .ascii "August"<0>
                .byte 0, 0, 0
aSeptember:     .ascii "September"<0>
aOctober:       .ascii "October"<0>
                .byte 0, 0
aNovember:      .ascii "November"<0>
                .byte    0
aDecember:      .ascii "December"<0>
                .byte 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Fazit

Das Gezeigte ist eine etwas altmodische Technik um Textstrings zu speichern. Man findet diesen Ansatz beispielsweise oft in Oracle RDBMS. Es ist schwer zu sagen, ob es sich für moderne Computer lohnt. Nichtsdestotrotz ist es ein gutes Beispiel für Arrays und hat daher seine Berechtigung in diesem Buch.

1.20.8 Fazit

Ein Array ist eine Ansammlung von Werten, die im Speicher nebeneinander angeordnet sind.

Dies gilt für alle Elementtypen und sogar für Structs.

Der Zugriff auf ein spezielles Element des Arrays entspricht lediglich eine Berechnung von dessen Adresse.

1.20.9 Übungen

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.21 Manipulieren einzelner Bits

Eine Menge Funktionen definiert ihre Eingabeargumente als Flags in Bitfields.

Natürlich können diese auch durch Variablen von Typ *bool* ersetzt werden; das wäre jedoch umständlicher als nötig.

1.21.1 Prüfen bestimmter Bits

x86

Win32 API Beipsiel:

```
    HANDLE fh;

    fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ↙
↘ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

We get (MSVC 2010):

Listing 1.239: MSVC 2010

```
    push    0
    push    128             ; 00000080H
    push    4
    push    0
    push    1
    push    -1073741824     ; c0000000H
    push    OFFSET $SG78813
    call    DWORD PTR __imp__CreateFileA@28
    mov     DWORD PTR _fh$[ebp], eax
```

Schauen wir uns WinNT.h genauer an:

Listing 1.240: WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Alles eindeutig beschrieben: GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000. Dieser Wert wird als zweites Argument für die Funktion `CreateFile()`[128] verwendet.

Wie würde `CreateFile()` diese Flags überprüfen? Wenn wir und die KERNEL32.DLL in Windows XP SP3 x86 anschauen, finden wir dieses Codefragment in `CreateFileW`:

Listing 1.241: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429     test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D     mov     [ebp+var_8], 1
.text:7C83D434     jz      short loc_7C83D417
.text:7C83D436     jmp     loc_7C810817
```

Wir finden hier den TEST Befehl, aber dieser nimmt nicht das ganze zweite Argument, sonder nur das MSB (`ebp+dwDesiredAccess+3`) und prüft es auf das Flag `0x40` (welches hier dem `GENERIC_WRITE` Flag entspricht).

TEST ist prinzipiell der gleiche Befehl wie AND, aber das Ergebnis wird nicht gespeichert. (Erinnern wir uns, dass CMP das gleiche macht wie SUB, aber auch ohne das Ergebnis zu speichern (1.9.4 on page 96)).

Die Logik dieses Codefragments ist die folgende:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

Wennn der AND Befehl dieses Bit hinterlässt, wird das ZF Flag gelöscht und der bedingte Sprung JZ wird nicht ausgeführt. Der bedingte Sprung wird nur dann ausgeführt, wenn das Bit `0x40000000` in der Variable `dwDesiredAccess` fehlt —dann ist das Ergebnis von AND 0, ZF wird gesetzt und der bedingte Sprung wird ausgeführt.

Schauen wir es uns mit GCC 4.4.1 unter Linux an:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;

        handle=open ("file", O_RDWR | O_CREAT);
};
```

Wir erhalten folgenden Code:

[128] msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx

Listing 1.242: GCC 4.4.1

```
                public main
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_1C], 42h
                mov     [esp+20h+var_20], offset aFile ; "file"
                call    _open
                mov     [esp+20h+var_4], eax
                leave
                retn
main            endp
```

Wenn wir uns die Funktion `open()` in der Bibliothek `libc.so.6` anschauen, gibt es nur einen syscall:

Listing 1.243: open() (libc.so.6)

```
.text:000BE69B     mov     edx, [esp+4+mode] ; mode
.text:000BE69F     mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3     mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7     mov     eax, 5
.text:000BE6AC     int     80h              ; LINUX - sys_open
```

Die Bitfields für `open()` werden also offenbar irgendwo im Linux Kernel geprüft.

Natürlich ist es ein Leichtes sowohl GLibc als auch den Quellcode des Linux Kernels herunterzuladen, aber wir wollen wir Sache ohne den Quellcode verstehen.

Wenn also in Linux 2.6 der syscall `sys_open` verwendet wird, wird die Kontrolle an `do_sys_open` übergeben und anschließend von dort aus—an die Funktion `do_filp_open()` (diese befindet sich im Verzeichnisbaum des Kernel-Quellcodes in `fs/namei.c`).

Neben der Übergabe von Argumenten über den Stack gibt es auch die Möglichkeit einige von ihnen direkt über Register zu übergeben. Dies wird auch fastcall (6.1.3 on page 582) genannt. Es ist schneller, da die CPU nicht auf den Stack im Speicher zugreifen muss, um die Funktionsargumente einzulesen. GCC kennt dafür die Option *regparm*[129], mithilfe derer es möglich ist, die Anzahl der Argumente anzugeben, die über Register übergeben werden sollen.

Der Linux 2.6 Kernel wird mit der Option `-mregparm=3` kompiliert [130] [131].

129 ohse.de/uwe/articles/gcc-attributes.html#func-regparm
130 kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f
131 Siehe auch die `arch/x86/include/asm/calling.h` Datei im Kernel Verzeichnisbaum

Dies bedeutet, dass die erste 3 Argumente über die Register EAX, EDX und ECX übergeben werden und der Rest über den Stack. Ist die Anzahl der Argumente kleiner als 3 wird natürlich nur ein Teil der genannten Register verwendet.

Laden wir also den Linux Kernel 2.6.31 herunter, kompilieren ihn in Ubuntu (make vmlinux) und öffnen ihn in IDA, so finden wir die Funktion do_filp_open(). Am Beginn derselben finden wir den folgenden Code (die Kommentare stammen vom Autor):

Listing 1.244: do_filp_open() (Linux Kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th argument)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1th argument)
                mov     [ebp+var_7C], edx ; pathname (2th argument)
                mov     [ebp+var_78], ecx ; open_flag (3th argument)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

GCC speichert die Werte der ersten drei Argumente auf dem lokalen Stack. Der Compiler würde diese Register ansonsten nicht verwenden und das wäre für den Register Allokator des Compilers nicht umsetzbar.

Finden wir dieses Codefragment:

Listing 1.245: do_filp_open() (Linux Kernel 2.6.31)

```
loc_C01EF684:          ; CODE XREF: do_filp_open+4F
                test    bl, 40h         ; O_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

0x40—enspricht dem O_CREAT Makro. open_flag wird auf Anwesenheit des 0x40 Bits hin überprüft und, wenn das Bit 1 ist, wird der folgende JNZ Befehl ausgelöst.

ARM

Das O_CREAT Bit wird im Linux Kernel 3.8.0 anders überprüft:

Listing 1.246: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                struct nameidata *nd, const struct open_flags *op, int ↙
    ↳ flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}


static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

So sieht der für den ARM mode kompilierte Kernel in IDA aus:

Listing 1.247: do_last() aus vmlinux (IDA)

```
...
.text:C0169EA8      MOV      R9, R3  ; R3 - (4th argument) open_flag
...
.text:C0169ED4      LDR      R6, [R9] ; R6 - open_flag
...
.text:C0169F68      TST      R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C      BNE      loc_C016A128
.text:C0169F70      LDR      R2, [R4,#0x10]
.text:C0169F74      ADD      R12, R4, #8
.text:C0169F78      LDR      R3, [R4,#0xC]
.text:C0169F7C      MOV      R0, R4
.text:C0169F80      STR      R12, [R11,#var_50]
.text:C0169F84      LDRB     R3, [R2,R3]
.text:C0169F88      MOV      R2, R8
.text:C0169F8C      CMP      R3, #0
```

```
.text:C0169F90      ORRNE     R1, R1, #3
.text:C0169F94      STRNE     R1, [R4,#0x24]
.text:C0169F98      ANDS      R3, R6, #0x200000
.text:C0169F9C      MOV       R1, R12
.text:C0169FA0      LDRNE     R3, [R4,#0x24]
.text:C0169FA4      ANDNE     R3, R3, #1
.text:C0169FA8      EORNE     R3, R3, #1
.text:C0169FAC      STR       R3, [R11,#var_54]
.text:C0169FB0      SUB       R3, R11, #-var_38
.text:C0169FB4      BL        lookup_fast
...
.text:C016A128 loc_C016A128   ; CODE XREF: do_last.isra.14+DC
.text:C016A128      MOV       R0, R4
.text:C016A12C      BL        complete_walk
...
```

TST ist analog zum Befehl TEST in x86. Wir können dies in diesem Codefragment daran erkennen, dass entweder lookup_fast() oder complete_walk() ausgeführt wird. Dies entspricht dem Quellcode der Funktion do_last(). Das Makro O_CREAT entspricht hier 0x40.

1.21.2 Setzen und löschen bestimmter Bits

Zum Beispiel:

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};
```

x86

Nicht optimierender MSVC

Wir erhalten folgenden Code: (MSVC 2010):

Listing 1.248: MSVC 2010

```
_rt$ = -4          ; size = 4
_a$ = 8            ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
    mov    ecx, DWORD PTR _rt$[ebp]
    or     ecx, 16384                ; 00004000H
    mov    DWORD PTR _rt$[ebp], ecx
    mov    edx, DWORD PTR _rt$[ebp]
    and    edx, -513                ; fffffdffH
    mov    DWORD PTR _rt$[ebp], edx
    mov    eax, DWORD PTR _rt$[ebp]
    mov    esp, ebp
    pop    ebp
    ret    0
_f  ENDP
```

Der OR Befehl setzt ein Bit auf einen Wert, während die übrigen ignoriert werden.

AND resettet ein Bit. Man kann sagen, dass AND einfach alle Bits bis auf eines kopiert. Tatsächlich werden im zweiten Operanden von AND nur die Bits gesetzt, die auch gespeichert werden müssen, lediglich das eine, das nicht kopiert werden soll (die 0 in der Bitmaske), wird nicht gesetzt. Auf diese Weise kann man sich die Logik des Befehls leichter merken.

OllyDbg

Untersuchen wir dieses Beispiel in OllyDbg. Schauen wir zunächst die binäre Form der zu verwendenden Konstanten an:

0x200 (0b00000000000000000001000000000) (d.h. das 10. Bit (vom ersten aus gezählt)).

0x200 ist invertiert 0xFFFFFDFF (0b11111111111111111110111111111).

0x4000 (0b00000000000000000100000000000000) (d.h. das 15. Bit).

Der Eingabewert ist: 0x12340678 (0b10010001101000000011001111000). Wir beobachten, wie der Wert geladen wird:

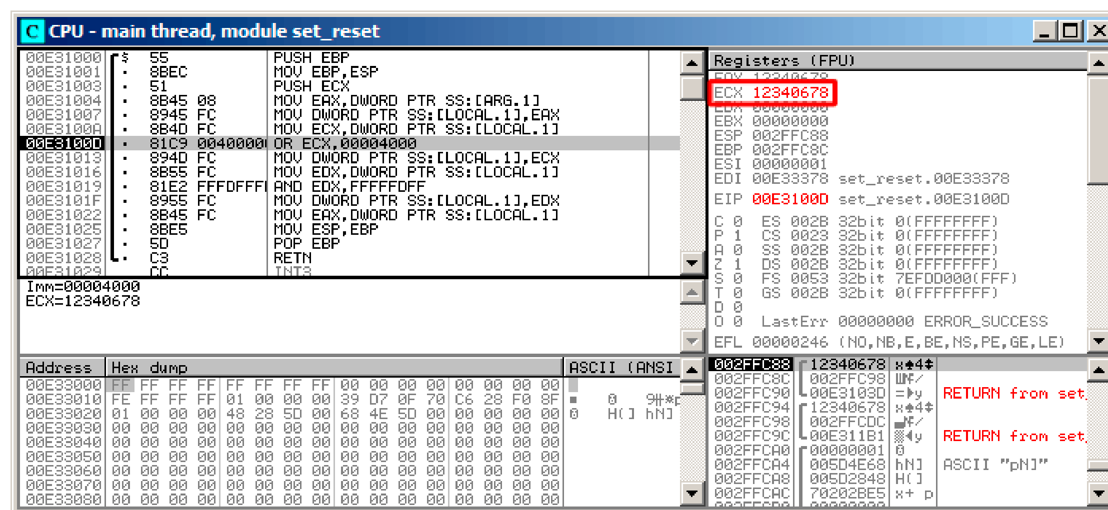


Abbildung 1.94: OllyDbg: der Wert wird nach ECX geladen

0R wurde ausgeführt:

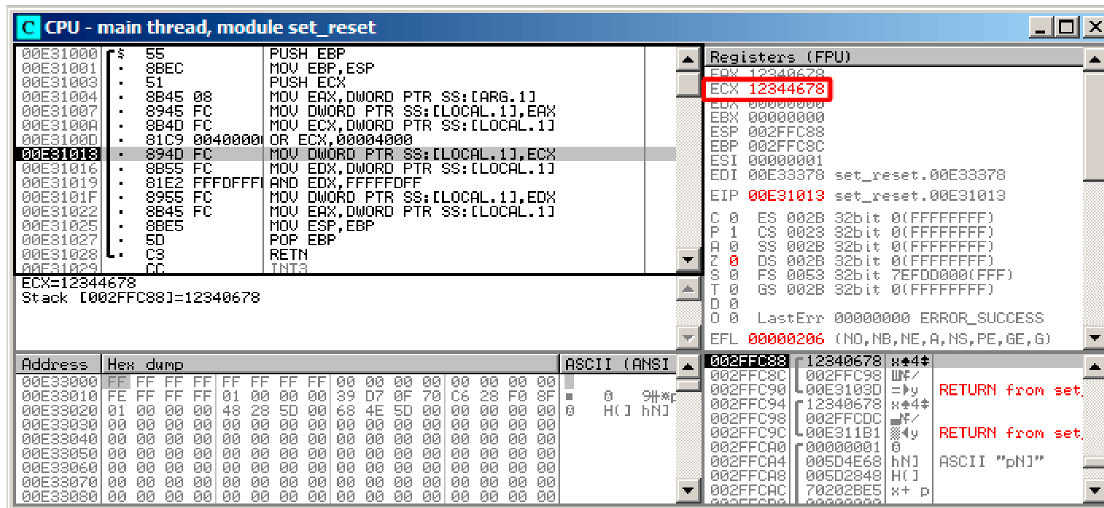


Abbildung 1.95: OllyDbg: 0R wurde ausgeführt

Das 15. Bit ist gesetzt: 0x12344678 (0b10010001101000100011001111000).

Der Wert wird erneut geladen (da der Compiler nicht optimiert hat):

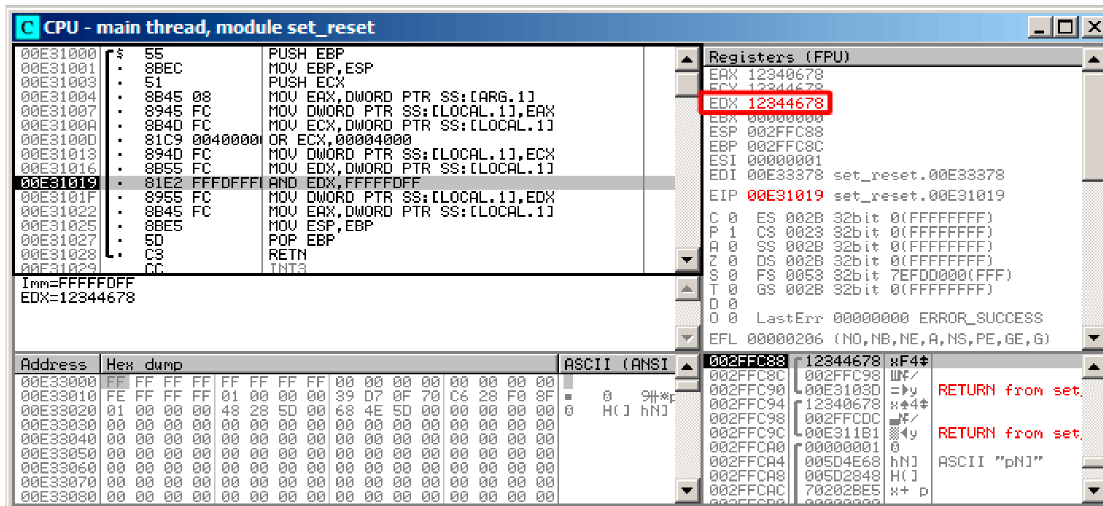


Abbildung 1.96: OllyDbg: der Wert wird erneut nach EDX geladen

AND wurde ausgeführt:

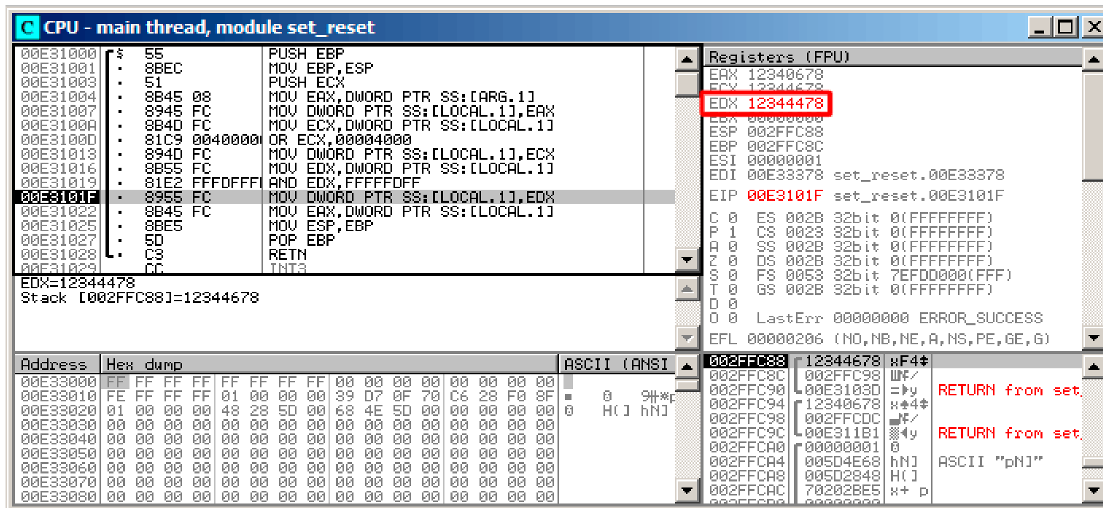


Abbildung 1.97: OllyDbg: AND wurde ausgeführt

Das 10. Bit wurde gelöscht (oder, mit anderen Worten: alle Bite außer dem 10. wurden stehengelassen) und das Endergebnis ist
0x12344478 (0b10010001101000100010001111000).

Optimierender MSVC

Wenn wir das Beispiel mit MSVC mit Optimierung (/Ox) kompilieren, ist der Code noch kürzer:

Listing 1.249: Optimierender MSVC

```
_a$ = 8                         ; size = 4
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513           ; fffffdffH
    or      eax, 16384          ; 00004000H
    ret     0
_f    ENDP
```

Nicht optimierender GCC

Untersuchen wir GCC 4.4.1 ohne Optimierung:

Listing 1.250: Nicht optimierender GCC

```
f                public f
                 proc near
```

```
var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
                and     [ebp+var_4], 0FFFFFDFFh
                mov     eax, [ebp+var_4]
                leave
                retn
f               endp
```

Obwohl es hier redundanten Code gibt, ist das Ergebnis kürzer als die MSVC Version ohne Optimierung.

Jetzt aktivieren wir die Optimierung von GCC mit `-O3`:

Optimierender GCC

Listing 1.251: Optimierender GCC

```
                public f
f               proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f               endp
```

Das Ergebnis ist noch kürzer. Man beachte, dass der Compiler mit dem EAX Regsiter über das Teilregister AH arbeitet—das ist der Teil vom 8. bis zum 15. Bit (jeweils einschließlich).

<table>
<tr><td colspan="8">Byte-Nummer:</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">RAX^{x64}</td></tr>
<tr><td colspan="4"></td><td colspan="4">EAX</td></tr>
<tr><td colspan="6"></td><td colspan="2">AX</td></tr>
<tr><td colspan="6"></td><td>AH</td><td>AL</td></tr>
</table>

Der 16-Bit CPU 8086 Akkumulator wurde AX getauft und bestand aus zwei 8-Bit-Hälften—AL (niederes Byte) und AH (höheres Byte). In 80386 wurden fast alle Regis-

ter auf 32 Bit erweitert und der Akkumulator wurde fortan EAX genannt, aber aus Kompatibilitätsgründen ist es immernoch möglich gezielt AX/AH/AL anzusprechen.

Da alle x86 CPUs Nachfolger der 16-Bit 8086 CPU sind, sind die älteren 16-Bit Opcodes kürzer als die neueren 32-Bit Opcodes. Aus diesem Gründ benötigt der Befehl `or ah, 40h` nur 3 Bytes. Logischer wäre es zwar, hier `or eax, 04000h` zu verwenden, aber dieser Befehl würde 5 oder sogar 6 Byte (falls das Register im ersten Operanden nicht EAX ist) verbrauchen.

Optimierender GCC und regparm

Es wäre noch kürzer, wenn man die Optimierung mit `-O3` anschaltet und `regparm=3` setzt.

Listing 1.252: Optimierender GCC

```
                public f
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

Das erste Argument ist schon nach EAX geladen worden, sodass es möglich ist, damit direkt weiterzuarbeiten. Bemerkenswert ist, dass sowohl der Funktionsprolog (`push ebp / mov ebp,esp`) als auch der Funktionsepilog (`pop ebp`) hier wegfallen können. GCC ist aber möglicherweise nicht gut genug um eine solche Code Optimierung hier durchzuführen. Auf jeden Fall sind solche kurzen Funktion am besten als *inline functions* (**??** on page ??) zu kennzeichnen.

ARM + Optimierender Keil 6/2013 (ARM Modus)

Listing 1.253: Optimierender Keil 6/2013 (ARM Modus)

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

Der Befehl BIC (*BItwise bit Clear*) dient zum Löschen spezifischer Bits. Er arbeitet wie ein AND Befehl mit invertierten Operanden. Er entspricht also einem NOT +AND Befehlspaar.

ORR bedeutet „logical or" und ist analog zu OR in x86.

So weit, so gut.

ARM + Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.254: Optimierender Keil 6/2013 (Thumb Modus)

```
01 21 89 03         MOVS    R1, 0x4000
08 43               ORRS    R0, R1
49 11               ASRS    R1, R1, #5   ; erzeuge 0x200 und speichere in R1
88 43               BICS    R0, R1
70 47               BX      LR
```

Es scheint, dass Keil enschieden hat, dass der Code im Thumn mode, der `0x200` statt `0x4000` verwendet, kompakter ist, als einer, der `0x200` in ein beliebiges Register schreibt.

Das ist der Grund dafür, dass dieser Wert mithilfe von ASRS (Arithmetische Linksverschiebung) als `0x4000` $\gg 5$ berechnet wird.

ARM + Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Listing 1.255: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
42 0C C0 E3         BIC             R0, R0, #0x4200
01 09 80 E3         ORR             R0, R0, #0x4000
1E FF 2F E1         BX              LR
```

Der Code, der von LLVM erzeugt wurde, könnte als Quellcode etwa wie folgt ausgesehen haben:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

Er macht genau das, was wir erwarten. Aber woher das `0x4200`? Möglicherweise handelt es sich um ein Artefakt des Optimierers von LLVM.

[132]. Möglicherweise also ein Fehler des Optimierers im Compiler; aber der erzeugte Code funktioniert trotzdem korrekt.

Mehr zu Compiler-Anomalien hier: (10.5 on page 667).

Optimierender Xcode 4.6.3 (LLVM) im Thumb mode erzeugt identischen Code.

ARM: Mehr zum Befehl BIC

Überarbeiten wir unser Beispiel ein wenig:

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

[132] Hier wurde der LLVM Build 2410.2.00 mit Apple Xcode 4.6.3 gebündelt

Jetzt erzeugt der optimierende Keil 5.03 im ARM mode folgenden Code:

```
f PROC
        BIC      r0,r0,#0x1000
        BIC      r0,r0,#0x234
        BX       lr
        ENDP
```

Es gibt zwei BIC Befehle, d.h. die Bits 0x1234 werden in zwei Durchgängen gelöscht.

Das liegt daran, dass es nicht möglich ist, 0x1234 in einem einzigen BIC Befehl zu kodieren; deshalb müssen 0x1000 und 0x234 getrennt werden.

ARM64: Optimierender GCC (Linaro) 4.9

Optimierender GCCcompiling für ARM64 kann den Befehl AND anstelle von BIC verwenden:

Listing 1.256: Optimierender GCC (Linaro) 4.9

```
f:
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        orr     w0, w0, 16384   ; 0x4000
        ret
```

ARM64: Nicht optimierender GCC (Linaro) 4.9

Nicht optimierender GCC erzeugt mehr redundanten Code; dieser funktiniert aber wie die optimierte Variante:

Listing 1.257: Nicht optimierender GCC (Linaro) 4.9

```
f:
        sub     sp, sp, #32
        str     w0, [sp,12]
        ldr     w0, [sp,12]
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        orr     w0, w0, 16384   ; 0x4000
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        and     w0, w0, -513    ; 0xFFFFFFFFFFFFFDFF
        str     w0, [sp,28]
        ldr     w0, [sp,28]
        add     sp, sp, 32
        ret
```

MIPS

Listing 1.258: Optimierender GCC 4.4.5 (IDA)

```
f:
```

```
; $a0=a
                ori     $a0, 0x4000
; $a0=a|0x4000
                li      $v0, 0xFFFFFDFF
                jr      $ra
                and     $v0, $a0, $v0
; am Ende gilt: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFFDFF
```

ORI ist natürlich ein OR Befehl. Das „I“ im Befehlsnamen bedeutet, dass der Wert in den Maschinencode eingebettet wird.

Danach finden wir AND. Hier kann nich tANDI verwendet werden, da es nicht möglich ist, die Zahl 0xFFFFFDFF in einen einzigen Befehl einzubetten, sodass der Compiler zunächst 0xFFFFFDFF in das Register $V0 lädt und dann ein AND erzeugt, das alle seine Eingabewerte aus den Registern entnimmt.

1.21.3 Verschiebungen

Bitverschiebungen sind in C/C++ mit den Befehlen ≪ und ≫ implementiert. Die x86 ISA verwendet die Befehle SHL (SHift Left) und SHR (SHift Right) zu diesem Zweck. Schiebebefehle werden oft bei der Division und Multiplikation mit Potenzen von 2 2^n (d.h. 1,2,4,8, etc.) verwendet: 1.18.1 on page 246, 1.18.2 on page 252.

Schiebebefehle sind auch wichtig, da sie oft für die Isolation einzelnes Bits oder für die Konstruktion von Werten aus mehreren einzelnen Bits verwendet werden.

1.21.4 Setzen und Löschen einzelner Bits: FPU Beispiele

In der folgenden Form werden die Bits in einem *float* gemäß IEEE 754 Format abgelegt:



(S — Vorzeichen)

Das Vorzeichen der Zahl ist im **MSB!**[133] kodiert. Wir fragen uns, ob es möglich ist, das Vorzeichen einer Fließkommazahl ohne FPU Befehle zu ändern.

```
#include <stdio.h>

float my_abs (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
        return *(float*)&tmp;
};

float set_sign (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
```

[133] **MSB!**

```
        return *(float*)&tmp;
};

float negate (float i)
{
        unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
        return *(float*)&tmp;
};

int main()
{
        printf ("my_abs():\n");
        printf ("%f\n", my_abs (123.456));
        printf ("%f\n", my_abs (-456.123));
        printf ("set_sign():\n");
        printf ("%f\n", set_sign (123.456));
        printf ("%f\n", set_sign (-456.123));
        printf ("negate():\n");
        printf ("%f\n", negate (123.456));
        printf ("%f\n", negate (-456.123));
};
```

Wir brauchen dieser Trickserei in C/C++ um von oder in *float* Werte ohne tatsächliche Konvertierung zu kopieren. Hier gibt es also drei Funktionen: my_abs() resettet **MSB!**; `set_sign()` setzt das **MSB!** und negate() kippt es.

XOR kann verwendet werden, um ein Bit zu kippen.

x86

Der Code ist einfach:

Listing 1.259: Optimierender MSVC 2012

```
_tmp$ = 8
_i$ = 8
_my_abs PROC
        and     DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
        or      DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
        xor     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
```

```
        fld     DWORD PTR _tmp$[esp-4]
        ret     0
_negate ENDP
```

Ein Eingabewert von Typ *float* wird vom Stack gelesen, aber wie ein Integerwert behandelt.

AND und OR resetten und setzen das gewünschte Bit und XOR kippt es. Schließlich wird der modifizierte Wert nach ST0 geladen, das Fließkommazahlen über dieses Register zurückgegeben werden.

Betrachten wir den optimierenden MSVC 2012 für x64:

Listing 1.260: Optimierender MSVC 2012 x64

```
tmp$ = 8
i$ = 8
my_abs  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btr     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
my_abs  ENDP
_TEXT   ENDS

tmp$ = 8
i$ = 8
set_sign PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        bts     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate  PROC
        movss   DWORD PTR [rsp+8], xmm0
        mov     eax, DWORD PTR i$[rsp]
        btc     eax, 31
        mov     DWORD PTR tmp$[rsp], eax
        movss   xmm0, DWORD PTR tmp$[rsp]
        ret     0
negate  ENDP
```

Der Eingabewert wird nach XMM0 übergeben und dann auf den lokalen Stack kopiert. Wir finden hier einige für uns neue Befehle: BTR, BTS und BTC.

Diese Befehle werden zum Resetten (BTR), Setzen (BTS) und Invertieren (oder Komplementieren: BTC) einzelner Bits verwendet. Das 31. Bit von 0 gezählt ist das **MSB!**.

Schließlich wird das Ergebnis nach XMM0 kopiert, da Fließkommazahlen in einer Win64 Umgebung über das Register XMM0 zurückgegeben werden.

MIPS

GCC 4.4.5 für MIPS erzeugt im Großen und Ganzen den gleichen Code:

Listing 1.261: Optimierender GCC 4.4.5 (IDA)

```
my_abs:
; hole 1 vom Koprozessor:
                mfc1    $v1, $f12
                li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; führe AND aus:
                and     $v0, $v1
; kopiere 1 zum Koprozessor 1:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot

set_sign:
; hole 1 vom Koprozessor:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; führe OR aus:
                or      $v0, $v1, $v0
; kopiere 1 zum Koprozessor:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot

negate:
; hole 1 vom Koprozessor:
                mfc1    $v0, $f12
                lui     $v1, 0x8000
; $v1=0x80000000
; do XOR:
                xor     $v0, $v1, $v0
; kopiere 1 zum Koprozessor:
                mtc1    $v0, $f0
; return
                jr      $ra
                or      $at, $zero ; branch delay slot
```

Ein einzelner LUI Befehl wird verwendet, um 0x80000000 in ein Register zu laden, den LUI löscht die niederen 16 Bits und da diese ohnehin Nullen in der Konstanten entsprechen genügt hier ein LUI ohne nachfolgendes ORI.

ARM

Optimierender Keil 6/2013 (ARM Modus)

Listing 1.262: Optimierender Keil 6/2013 (ARM Modus)

```
my_abs PROC
; lösche Bit:
        BIC      r0,r0,#0x80000000
        BX       lr
        ENDP

set_sign PROC
; führe OR aus:
        ORR      r0,r0,#0x80000000
        BX       lr
        ENDP

negate PROC
; führe XOR aus:
        EOR      r0,r0,#0x80000000
        BX       lr
        ENDP
```

So weit, so gut. ARM verfügt über den BIC Befehl, der ausdrücklich spezifizierte Bits löscht. EOR ist in ARM der Name des Befehls für XOR („exklusives OR").

Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.263: Optimierender Keil 6/2013 (Thumb Modus)

```
my_abs PROC
        LSLS     r0,r0,#1
; r0=i<<1
        LSRS     r0,r0,#1
; r0=(i<<1)>>1
        BX       lr
        ENDP

set_sign PROC
        MOVS     r1,#1
; r1=1
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        ORRS     r0,r0,r1
; r0=r0 | 0x80000000
        BX       lr
        ENDP

negate PROC
        MOVS     r1,#1
; r1=1
```

```
        LSLS     r1,r1,#31
; r1=1<<31=0x80000000
        EORS     r0,r0,r1
; r0=r0 ^ 0x80000000
        BX       lr
        ENDP
```

Thumb mode im ARM verwendet 16-Bit-Befehle und da in diesen nicht viele Daten kodiert werden können, wird hier ein MOVS/LSLS Befehlspaar benötigt, um die Konstante 0x80000000 zu konstruieren. Das Befehlspaar funktioniert wie folgt: $1 << 31 = 0x80000000$.

Der Code von my_abs ist seltsam und entspricht tatsächlich dem folgenden Ausdruck: $(i << 1) >> 1$. Dieser Ausdruck scheint zunächst bedeutungslos. Wenn aber $input << 1$ ausgeführt wird, befinden sich alle Bits an ihren korrekten Plätzen, nur dass das **MSB!** null ist, da alle neuen Bits aus, die durch den Schiebebefehl eingefügt werden, stets Nullen sind. Auf diese Weise löscht das Befehlspaar LSLS/LSRS das **MSB!**.

Optimierender GCC 4.6.3 (Raspberry Pi, ARM Modus)

Listing 1.264: Optimierender GCC 4.6.3 for Raspberry Pi (ARM Modus)

```
my_abs
; kopiere von S0 nach R2:
                FMRS    R2, S0
; lösche Bit:
                BIC     R3, R2, #0x80000000
; kopiere von R3 nach S0:
                FMSR    S0, R3
                BX      LR

set_sign
; kopiere von S0 nach R2:
                FMRS    R2, S0
; führe OR aus:
                ORR     R3, R2, #0x80000000
; kopiere von R3 nach S0:
                FMSR    S0, R3
                BX      LR

negate
; kopiere von S0 nach R2:
                FMRS    R2, S0
; führe ADD aus:
                ADD     R3, R2, #0x80000000
; kopiere von R3 nach S0:
                FMSR    S0, R3
                BX      LR
```

Lassen wir den Raspberry Pi Linux in QEMU laufen und emulieren eine ARM FPU, dann werden hier S-Register anstelle der R-Register für den Umgang mit Fließkommazahlen verwendet.

Der Befehl FMRS kopiert Daten von GPR zur FPU und zurück `my_abs()` und `set_sign()` sehen wie erwartet aus, aber was ist mit `negate()`? Warum wird hier ADD anstelle von XOR verwendet?

Es ist auf den ersten Blick schwer zu glauben, aber der Befehl `ADD register, 0x80000000` entspricht
`XOR register, 0x80000000`. Erinnern wir uns an das Ziel des Befehls: Das Ziel ist es, das **MSB!** zu invertieren, also kümmern wir uns zunächst nicht um den XOR Befehl. Aus der Schulmathematik wissen wir, dass die Addition von Werten wie z.B. 1000 die letzten drei Stellen einer Zahl nicht verändert. Zum Beispiel gilt: $1234567 + 10000 = 1244567$ (die letzten vier Stellen können sich nicht verändern).

Hier arbeiten wir mit Binärzahlen und
0x80000000 ist 0b100000000000000000000000000000000, d.h. hier sind nur das höchste Bit gesetzt.

Eine Addition von 0x800000000 zu einem anderen Werte kann also nie die niederen 31 Bit verändern, sondern nur das **MSB!**. Addieren wir 1 zu 0, erhalten wir 1.

Addieren wir 1 zu 1, erhalten wir 0b10 in binär, aber das 32. Bit (von 0 gezählt) wird fallengelassen, da unsere Register eine Breite von 32 Bit haben, sodass das Ergebnis 0 ist. Deshalb kann in diesem Fall XOR durch ADD ersetzt werden.

Es ist schwer nachzuvollziehen, warum GCC diese Ersetzung vorgenommen hat, aber sie funktioniert tadellos.

1.21.5 Gesetzte Bits zählen

Hier ist ein einfaches Beispiel einer Funktion, die die Anzahl der gesetzten Bits in einem Eingabewert zählt.

Diese Operation wird auch „population count“[134] genannt.

```
#include <stdio.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
```

[134] moderne x86 CPUs (die SSE4 unterstützen) haben zu diesem Zweck sogar einen eigenen POPCNT Befehl

```
    f(0x12345678); // test
};
```

In dieser Schleife wird der Wert von i schrittweise von 0 bis 31 erhöht, sodass der Ausdruck $1 \ll i$ von 1 bis `0x80000000` zählt. In natürlicher Sprache würden wir diese Operation als *verschiebe 1 um n Bits nach links* beschreiben. Mit anderen Worten: Der Ausdruck $1 \ll i$ erzeugt alle möglichen Bitpositionen in einer 32-Bit-Zahl. Das freie Bit auf der rechten Seite wird jeweils gelöscht.

Hier ist eine Tabelle mit allen Werten von $1 \ll i$ für $i = 0 \dots 31$:

C/C++ Ausdruck	Zweierpotenz	Dezimalzahl	Hexadezimalzahl
$1 \ll 0$	2^0	1	1
$1 \ll 1$	2^1	2	2
$1 \ll 2$	2^2	4	4
$1 \ll 3$	2^3	8	8
$1 \ll 4$	2^4	16	0x10
$1 \ll 5$	2^5	32	0x20
$1 \ll 6$	2^6	64	0x40
$1 \ll 7$	2^7	128	0x80
$1 \ll 8$	2^8	256	0x100
$1 \ll 9$	2^9	512	0x200
$1 \ll 10$	2^{10}	1024	0x400
$1 \ll 11$	2^{11}	2048	0x800
$1 \ll 12$	2^{12}	4096	0x1000
$1 \ll 13$	2^{13}	8192	0x2000
$1 \ll 14$	2^{14}	16384	0x4000
$1 \ll 15$	2^{15}	32768	0x8000
$1 \ll 16$	2^{16}	65536	0x10000
$1 \ll 17$	2^{17}	131072	0x20000
$1 \ll 18$	2^{18}	262144	0x40000
$1 \ll 19$	2^{19}	524288	0x80000
$1 \ll 20$	2^{20}	1048576	0x100000
$1 \ll 21$	2^{21}	2097152	0x200000
$1 \ll 22$	2^{22}	4194304	0x400000
$1 \ll 23$	2^{23}	8388608	0x800000
$1 \ll 24$	2^{24}	16777216	0x1000000
$1 \ll 25$	2^{25}	33554432	0x2000000
$1 \ll 26$	2^{26}	67108864	0x4000000
$1 \ll 27$	2^{27}	134217728	0x8000000
$1 \ll 28$	2^{28}	268435456	0x10000000
$1 \ll 29$	2^{29}	536870912	0x20000000
$1 \ll 30$	2^{30}	1073741824	0x40000000
$1 \ll 31$	2^{31}	2147483648	0x80000000

Diese Konstanten (Bitmasken) tauchen im Code oft auf und ein Reverse Engineer muss in der Lage sein, sie schnell und sicher zu erkennen.

Es dazu jedoch nicht notwendig, die Dezimalzahlen (Zweierpotenzen) größer 65535 auswendig zu kennen. Die hexadezimalen Zahlen sind leicht zu merken.

Die Konstanten werden häufig verwendet um Flags einzelnen Bits zuzuordnen. Hier ist zum Beispiel ein Auszug aus `ssl_private.h` aus dem Quellcode von Apache 2.4.6:

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE           (0)
#define SSL_OPT_RELSET         (1<<0)
#define SSL_OPT_STDENVVARS     (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH  (1<<4)
#define SSL_OPT_STRICTREQUIRE  (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Zurück zu unserem Beispiel.

Das Makro IS_SET prüft auf Anwesenheit von Bits in a.

Das Makro IS_SET entspricht dabei dem logischen (*AND*) und gibt 0 zurück, wenn das entsprechende Bit nicht gesetzt ist, oder die Bitmaske, wenn das Bit gesetzt ist. Der Operator *if()* wird in C/C++ ausgeführt, wenn der boolesche Ausdruck nicht null ist (er könnte sogar 123456 sein), weshalb es meistens richtig funktioniert.

x86

MSVC

Kompilieren wir (MSVC 2010):

Listing 1.265: MSVC 2010

```
_rt$ = -8            ; size = 4
_i$ = -4             ; size = 4
_a$ = 8              ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]   ; erhöhe i
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; Schleife beendet?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
```

```
    and     edx, DWORD PTR _a$[ebp]
    je      SHORT $LN1@f             ; was das Ergebnis des AND Befehls 0?
                                     ; dann überspringe die nächsten Befehle
    mov     eax, DWORD PTR _rt$[ebp] ; nein: ungleich 0
    add     eax, 1                   ; erhöhe rt
    mov     DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp     SHORT $LN3@f
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

Betrachten wir dieses Beispiel in OllyDbg. Sei der Eingabewert dabei 0x12345678. Für $i = 1$ sehen wir, wie i nach ECX geladen wird:

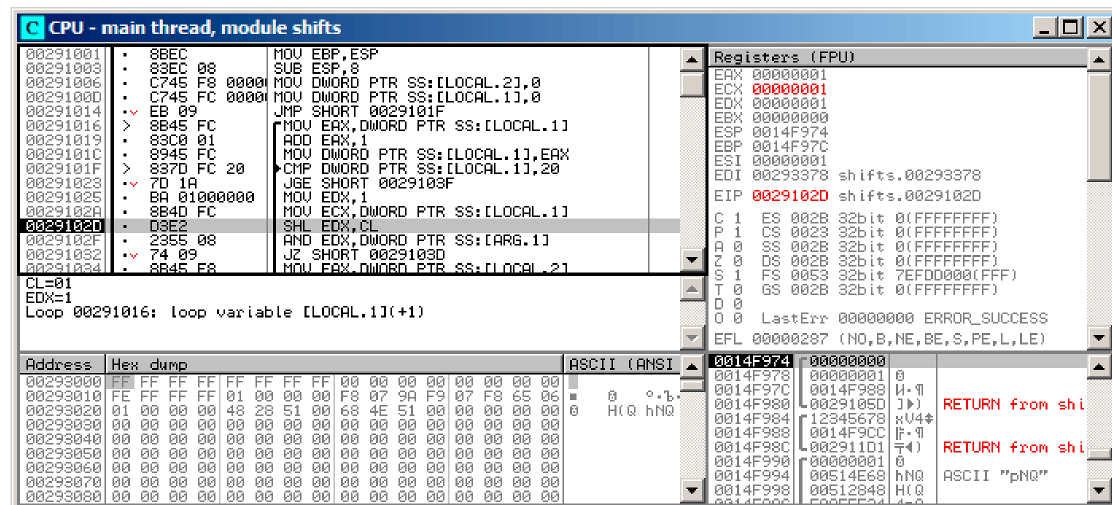


Abbildung 1.98: OllyDbg: $i = 1$, i wird nach ECX geladen

EDX ist 1. SHL wird jetzt ausgeführt.

SHL wurde ausgeführt:

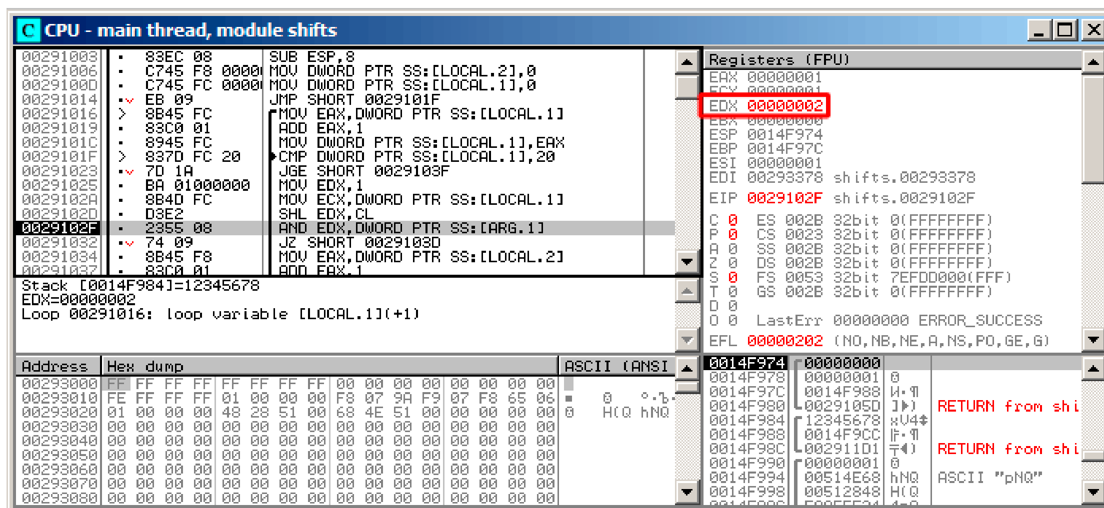


Abbildung 1.99: OllyDbg: $i = 1$, EDX $=1 \ll 1 = 2$

EDX enthält $1 \ll 1$ (oder 2). Hierbei handelt es sich um eine Bitmaske.

AND setzt ZF auf 1, was bedeutet, dass der Eingabewert (0x12345678) mit 2 verUNDet wird. Das Ergebnis ist 0:

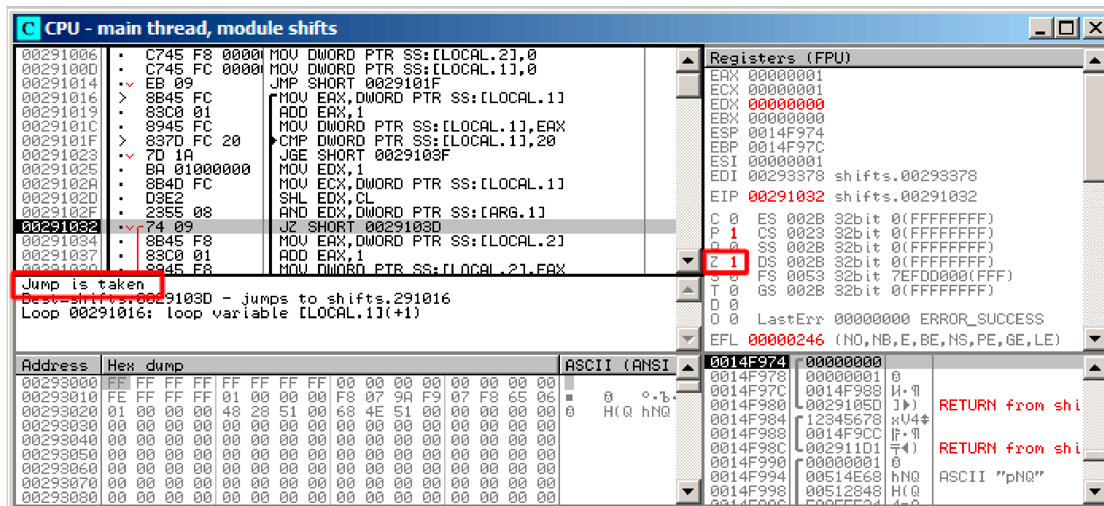


Abbildung 1.100: OllyDbg: $i = 1$, ist hier das Bit im Eingabewert gesetzt? Nein. (ZF =1)

Es gibt hier also kein entsprechendes Bit im Eingabewert.

Das Codestück, welches den Zähler erhöht, wird also nicht ausgeführt: Der JZ Befehl überspringt es.

Verfolgen wir den Ablauf ein bisschen weiter bis i den Wert 4 hat. SHL wird jetzt ausgeführt:

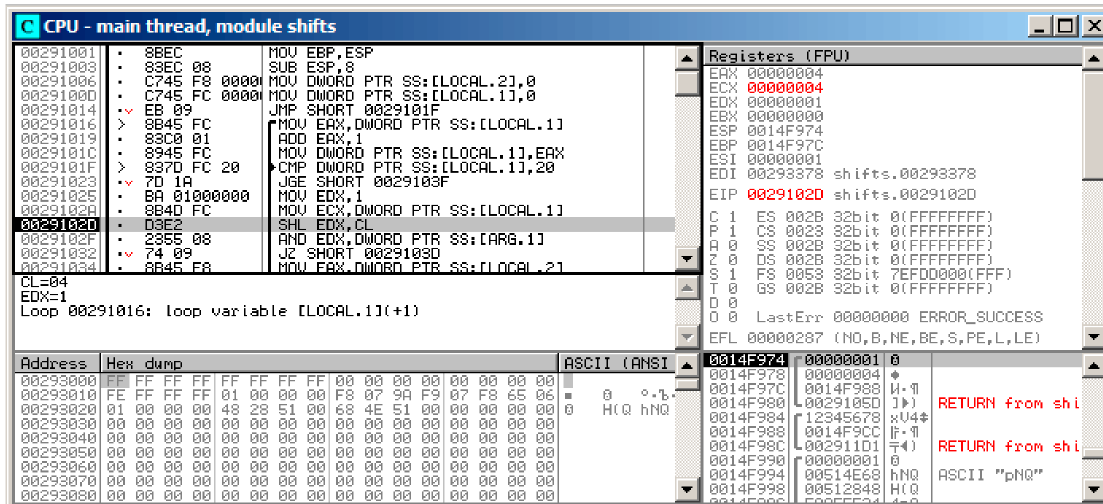


Abbildung 1.101: OllyDbg: $i = 4$, i wird nach ECX geladen

EDX $=1 \ll 4$ (oder `0x10` oder 16):

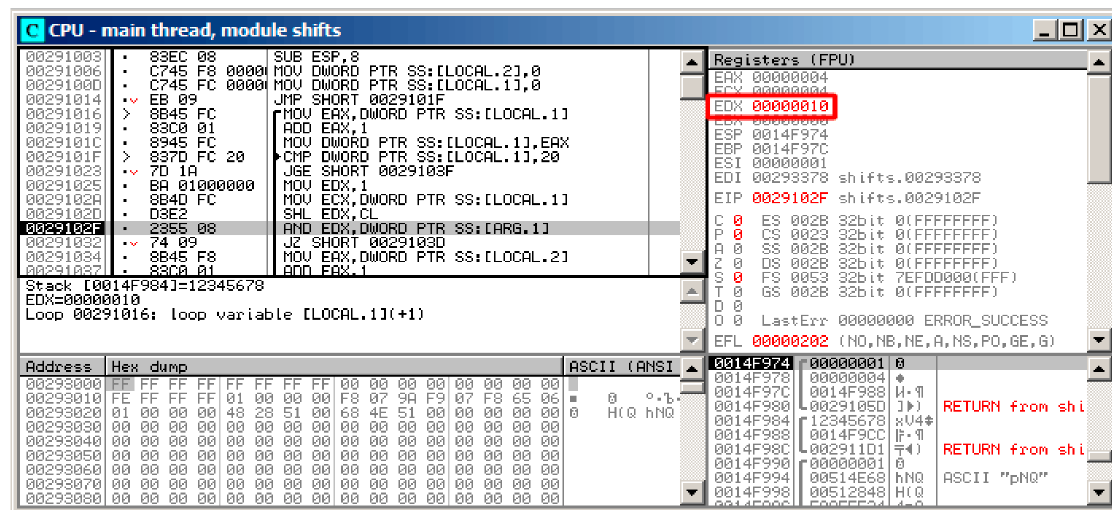


Abbildung 1.102: OllyDbg: $i = 4$, EDX $=1 \ll 4 = 0x10$

Hierbei handelt es sich um eine weitere Bitmaske.

AND wird ausgeführt:

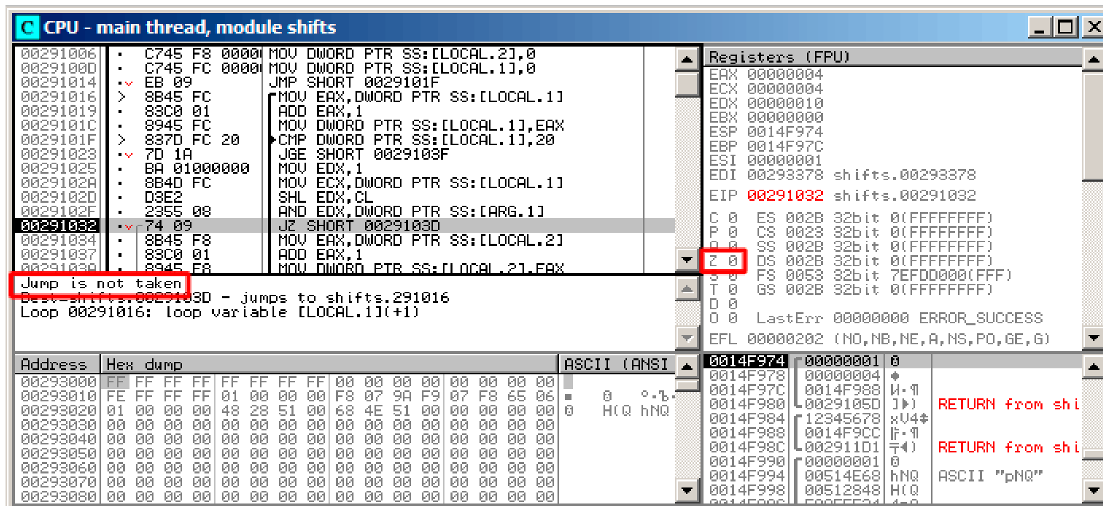


Abbildung 1.103: OllyDbg: $i = 4$, ist hier das Bit im Eingabewert gesetzt? Ja. (ZF =0)

ZF ist 0 , da das Bit im Eingabewert gesetzt ist.
Tatsächlich gilt `0x12345678 & 0x10 = 0x10`.

Das Bit wird gezählt: der Sprung wird nicht ausgeführt und der Zähler wird erhöht.

Die Funktion liefert den Wert 13 zurück. Dies entspricht der Anzahl der in der binären Darstellung von `0x12345678` gesetzten Bits.

GCC

Kompilieren wir das Beispiel mit GCC 4.4.1:

Listing 1.266: GCC 4.4.1

```
                public f
f               proc near

rt              = dword ptr -0Ch
i               = dword ptr -8
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483EF
loc_80483D0:
```

```
                mov     eax, [ebp+i]
                mov     edx, 1
                mov     ebx, edx
                mov     ecx, eax
                shl     ebx, cl
                mov     eax, ebx
                and     eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_80483EB
                add     [ebp+rt], 1
loc_80483EB:
                add     [ebp+i], 1
loc_80483EF:
                cmp     [ebp+i], 1Fh
                jle     short loc_80483D0
                mov     eax, [ebp+rt]
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn
f               endp
```

x64

Verändern wir das Beispiel ein wenig um es auf 64 Bit zu erweitern:

```
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)       ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};
```

Nicht optimierender GCC 4.8.2

So weit, so einfach.

Listing 1.267: Nicht optimierender GCC 4.8.2

```
f:
        push    rbp
```

```
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi ; a
        mov     DWORD PTR [rbp-12], 0   ; rt=0
        mov     QWORD PTR [rbp-8], 0    ; i=0
        jmp     .L2
.L4:
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; ist das letzte Bit 0?
; überspringe nächsten ADD Befehl, wenn es so war.
        je      .L3
        add     DWORD PTR [rbp-12], 1   ; rt++
.L3:
        add     QWORD PTR [rbp-8], 1    ; i++
.L2:
        cmp     QWORD PTR [rbp-8], 63   ; i<63?
        jbe     .L4                     ; springe zum Beginn der Schleife, falls wahr
        mov     eax, DWORD PTR [rbp-12] ; return rt
        pop     rbp
        ret
```

Optimierender GCC 4.8.2

Listing 1.268: Optimierender GCC 4.8.2

```
f:
        xor     eax, eax        ; rt liegt in EAX
        xor     ecx, ecx        ; i liegt in ECX
.L3:
        mov     rsi, rdi        ; lade Eingabewert
        lea     edx, [rax+1]    ; EDX=EAX+1
; EDX enthält hier eine neue Version von rt,
; diese wird nach rt geschrieben, falls das letzte Bit 1 ist
        shr     rsi, cl         ; RSI=RSI>>CL
        and     esi, 1          ; ESI=ESI&1
; das letzte Bit ist 1? Falls ja, schreibe neue Version von rt nach EAX
        cmovne  eax, edx
        add     rcx, 1          ; RCX++
        cmp     rcx, 64
        jne     .L3
        rep ret                 ; AKA fatret
```

Dieser Code ist kürzer, birgt aber eine Besonderheit.

In allen bisher betrachteten Beispieln haben wir den Wert von „rt" nach dem Vergleich mit einem speziellen Bit erhöht, aber dieser Code erhöht „rt" vorher (Zeile 6) und schreibt den neuen Wert in das Register EDX. Dadurch überträgt der Befehl CMOVNE[135] (der ein Synonym für CMOVNZ[136] ist) den neuen Wert von „rt" durch Verschieben des Wertes in EDX (vorgeschlagener Wert von „rt") nach EAX („aktueller Wert von rt"). Der in EAX befindliche Wert wird schließlich zurückgegeben.

Deshalb wird die Erhöhung des Zählers in jedem Durchlauf der Schleife durchgeführt, d.h. 64 mal, ohne dass eine Abhängigkeit vom Eingabewert besteht.

Der Vorteil dieses Code ist, dass er nur einen bedingten Sprung enthält (am Ende der Schleife) anstatt zwei Sprüngen (Überspringen des Erhöhens von „rt" und Ende der Schleife). Der Code ist somit auf modernen CPUs mit Branch Pedictors möglicherweise schneller:**??** on page ??.

Der letzte Befehl hier ist REP RET (Opcode F3 C3), der von MSVC auch FATRET genannt wird. Hierbei handelt es sich um eine optimierte Version von RET, die von ARM bevorzugt am Ende der Funktion verwendet wird, wenn RET direkt nach einem bedingten Sprung folg:. [*Software Optimization Guide for AMD Family 16h Processors*, (2013)p.15] [137].

Optimierender MSVC 2010

Listing 1.269: Optimierender MSVC 2010

```
a$ = 8
f       PROC
; RCX = input value
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+64]
; R8D=64
        npad    5
$LL4@f:
        test    rdx, rcx
; Eingabewert enthält kein solches Bit?
; dann überspringen nächsten INC Befehl.
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

[135]Conditional MOVe if Not Equal
[136]Conditional MOVe if Not Zero
[137]Mehr Informationen dazu: http://repzret.org/p/repzret/

Hier wird der Befehl ROL anstelle von SHL verwendet, welches einer „Linksrotation“ anstatt einer „Linksverschiebung“ entspricht. In diesem Beispiel entspricht ROL einem SHL.

Für mehr Informationen zu Rotationsbefehlen siehe: **??** on page ??.

R8 zählt hier von 64 auf 0 herunter. Dies entspricht dem invertierten i.

Hier ist eine Tabelle einiger Register während der Ausführung des Programms:

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

Am Ende finden wir den Befehl FATRET, der hier schon erklärt wurde:1.21.5 on the previous page.

Optimierender MSVC 2012

Listing 1.270: Optimierender MSVC 2012

```
a$ = 8
f       PROC
; RCX = Eingabewert
        xor     eax, eax
        mov     edx, 1
        lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
        npad    5
$LL4@f:
; übergib 1 --------------------------
        test    rdx, rcx
        je      SHORT $LN3@f
        inc     eax     ; rt++
$LN3@f:
        rol     rdx, 1  ; RDX=RDX<<1
; ------------------------------------
; übergib 2 --------------------------
        test    rdx, rcx
        je      SHORT $LN11@f
        inc     eax     ; rt++
$LN11@f:
        rol     rdx, 1  ; RDX=RDX<<1
; ------------------------------------
        dec     r8      ; R8--
        jne     SHORT $LL4@f
        fatret  0
f       ENDP
```

Der optimierende MSVC 2012 erzeugt fast den gleichen Code wie MSVC 2012, generiert aber aus irgendeinem Grund zwei identischen Rümpfe für die Schleifen und die Schleife zählt nun bis 32 anstatt 64.

Ehrlich gesagt, kann man nicht genau erklären warum. Es könnte sich um einen Optimierungstrick handeln. Vielleicht ist es für den Rumpf der Schleife besser ein wenig länger zu sein.

Trotzdem ist solcher Code relevant um zu zeigen, dass der Output des Compilers manchmal sehr merkwürdig und unlogisch sein kann und dennoch tadellos funktioniert.

ARM + Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

Listing 1.271: Optimierender Xcode 4.6.3 (LLVM) (ARM Modus)

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; setze Flags entsprechend R1 &
    (R2 << R3)
                (R2<<R3) ADD    R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; wenn ZF Flag von TST gelöscht
    wurde, dann R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST entspricht dem Befehl TEST in x86.

Wie bereits in (**??** on page ??) besprochen gibt es zwei verschiedene Schiebebefehle im ARM mode. Zusätzlich gibt es aber noch die Suffixe LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) und RRX (*Rotate Right with Extend*), die an Befehle wie `MOV`, TST, `CMP`, `ADD`, `SUB`, RSB[138] angehängt werden können.

Diese Suffixe legen fest, wie und um wie viele Bits der zweite Operand verschoben werden soll.

Dadurch entspricht der Befehl „`TST R1, R2,LSL R3`“ hier $R1 \land (R2 \ll R3)$.

ARM + Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

Fast das gleiche, aber hier werden zwei `LSL.W`/TST Befehle anstelle eines einzelnen TST verwendet, da es im Thumb mode nicht möglich ist, den Suffix LSL direkt in TST zu definieren.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
```

[138] Diese Instruktionen werden auch „Datenverarbeitungsinstruktionen“ genannt

```
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
                BNE             loc_2F7A
                BX              LR
```

ARM64 + Optimierender GCC 4.9

Betrachten wir ein 64-Bit-Beispiel, das wir bereits kennen:1.21.5 on page 386.

Listing 1.272: Optimierender GCC (Linaro) 4.8

```
f:
        mov     w2, 0           ; rt=0
        mov     x5, 1
        mov     w1, w2
.L2:
        lsl     x4, x5, x1      ; w4 = w5<<w1 = 1<<i
        add     w3, w2, 1       ; new_rt=rt+1
        tst     x4, x0          ; (1<<i) & a
        add     w1, w1, 1       ; i++
; war das Ergebnis von TST ungleich null?
; dann w2=w3 oder rt=new_rt.
; sonst: w2=w2 oder rt=rt (leeres Befehl)
        csel    w2, w3, w2, ne
        cmp     w1, 64          ; i<64?
        bne     .L2             ; yes
        mov     w0, w2          ; return rt
        ret
```

Das Ergebnis ist ähnlich dem was GCC für x64 erzeugt:1.268 on page 387.

Der Befehl CSEL steht für „Conditional SELect“. Er wählt eine von zwei Variablen abhängig von den durch TST gesetzen Flags aus und kopiert deren Wert nach W2, wo die Variable „rt“ gespeichert wird.

ARM64 + Nicht optimierender GCC 4.9

Wieder werden wir hier mit dem bereits bekannten 64-Bit-Beispiel arbeiten: 1.21.5 on page 386. Der Code ist umfangreicher als gewöhnlich.

Listing 1.273: Nicht optimierender GCC (Linaro) 4.8

```
f:
        sub     sp, sp, #32
        str     x0, [sp,8]      ; speichere "a" in der Register Save Area
        str     wzr, [sp,24]    ; rt=0
        str     wzr, [sp,28]    ; i=0
```

```
        b       .L2
.L4:
        ldr     w0, [sp,28]
        mov     x1, 1
        lsl     x0, x1, x0      ; X0 = X1<<X0 = 1<<i
        mov     x1, x0
; X1 = 1<<i
        ldr     x0, [sp,8]
; X0 = a
        and     x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; enthält X0 null? Dann springe zu .L3, lasse "rt" Inkrement aus
        cmp     x0, xzr
        beq     .L3
; rt++
        ldr     w0, [sp,24]
        add     w0, w0, 1
        str     w0, [sp,24]
.L3:
; i++
        ldr     w0, [sp,28]
        add     w0, w0, 1
        str     w0, [sp,28]
.L2:
; i<=63? Dann springe zu .L4
        ldr     w0, [sp,28]
        cmp     w0, 63
        ble     .L4
; return rt
        ldr     w0, [sp,24]
        add     sp, sp, 32
        ret
```

MIPS

Nicht optimierender GCC

Listing 1.274: Nicht optimierender GCC 4.4.5 (IDA)

```
f:
; IDA vergibt keine Variablennamen, diese wurden manuell hinzugefügt:
rt              = -0x10
i               = -0xC
var_4           = -4
a               =  0

                addiu   $sp, -0x18
                sw      $fp, 0x18+var_4($sp)
                move    $fp, $sp
                sw      $a0, 0x18+a($fp)
; initialisiere rt und i mit 0:
                sw      $zero, 0x18+rt($fp)
```

```
                sw      $zero, 0x18+i($fp)
; Springe, um Schleifenbedingung zu prüfen:
                b       loc_68
                or      $at, $zero ; branch delay slot, NOP

loc_20:
                li      $v1, 1
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; lade delay slot, NOP
                sllv    $v0, $v1, $v0
; $v0 = 1<<i
                move    $v1, $v0
                lw      $v0, 0x18+a($fp)
                or      $at, $zero ; lade delay slot, NOP
                and     $v0, $v1, $v0
; $v0 = a & (1<<i)
; ist a & (1<<i) gleich null? Dann springe zu loc_58:
                beqz    $v0, loc_58
                or      $at, $zero
; kein Sprung, d.h. a & (1<<i)!=0, also erhöhe "rt":
                lw      $v0, 0x18+rt($fp)
                or      $at, $zero ; lade delay slot, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+rt($fp)

loc_58:
; erhöhe i:
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; lade delay slot, NOP
                addiu   $v0, 1
                sw      $v0, 0x18+i($fp)

loc_68:
; lade i und vergleiche mit 0x20 (32).
; springe zu loc_20 , falls kleiner 0x20 (32):
                lw      $v0, 0x18+i($fp)
                or      $at, $zero ; lade delay slot, NOP
                slti    $v0, 0x20  # ' '
                bnez    $v0, loc_20
                or      $at, $zero ; branch delay slot, NOP
; Funktionsepilog. Wert rt wird zurückgegeben:
                lw      $v0, 0x18+rt($fp)
                move    $sp, $fp   ; lade delay slot
                lw      $fp, 0x18+var_4($sp)
                addiu   $sp, 0x18  ; lade delay slot
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

Umständlich: alle lokalen Variablen liegen auf dem lokalen Stack und werden bei jedem Zugriff neu geladen.

Der Befehl SLLV bedeutet „Shift Word Left Logical Variable“. Er unterscheidet sich von SLL nur dadurch, dass die Anzahl der Verschiebungen im SLL Befehl kodiert sind

(und dadurch nicht veränderbar). SLLV hingegen erhält die Anzahl der Verschiebungen aus einem Register.

Optimierender GCC

Hier ist es knapper gehalten. Warum aber gibt es zwei Schiebebefehle anstatt eines?

Es ist möglich, den ersten SLLV Befehl durch einen unbedingte Verzweigungsbefehl zu ersetzen, der direkt zum zweiten SLLV springt. Das zieht aber einen zweiten Verzweigungsbefehl nach sich und es ist stets vorteilhaft sich dieser zu entledigen:**??** on page ??.

Listing 1.275: Optimierender GCC 4.4.5 (IDA)

```
f:
; $a0=a
; rt bleibt in $v0:
                move    $v0, $zero
; i bleibt in $v1:
                move    $v1, $zero
                li      $t0, 1
                li      $a3, 32
                sllv    $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14:
                and     $a1, $a0
; $a1 = a&(1<<i)
; erhöhe i:
                addiu   $v1, 1
; springe zu loc_28 , falls a&(1<<i)==0 und erhöhe rt:
                beqz    $a1, loc_28
                addiu   $a2, $v0, 1
; wenn BEQZ nicht ausgelöst wurde, speichere neues rt nach $v0:
                move    $v0, $a2

loc_28:
; falls i!=32, springe zu loc_14 und bereite nächsten verschobenen Wert vor:
                bne     $v1, $a3, loc_14
                sllv    $a1, $t0, $v1
; return
                jr      $ra
                or      $at, $zero ; branch delay slot, NOP
```

1.21.6 Fazit

Analog zu den Schiebebefehlen $\ll$ und $\gg$ in C/C++ gibt es in x86 die Befehle SHR/SHL (für vorzeichenlose Werte) und SAR/SHL (für vorzeichenbehaftete Werte).

Die Schiebebefehle in ARM sind LSR/LSL (für vorzeichenlose Werte) und ASR/LSL (für vorzeichenbehaftete Werte).

Es sind bei manchen Befehlen auch mögliche Suffixe für die Verschiebung anzuhängen (diese heiße „data processing instructions“).

Prüfung auf spezifisches Bit (zur Compilezeit bekannt)

Prüfung, ob das Bit 0b10000000 (0x40) sich im Registerwert befindet:

Listing 1.276: C/C++

```
if (input&0x40)
    ...
```

Listing 1.277: x86

```
TEST REG, 40h
JNZ is_set
; Bit ist nicht gesetzt
```

Listing 1.278: x86

```
TEST REG, 40h
JZ is_cleared
; Bit ist gesetzt
```

Listing 1.279: ARM (ARM Modus)

```
TST REG, #0x40
BNE is_set
; Bit ist nicht gesetzt
```

Manchmal wird AND anstelle von TEST verwendet, aber die gesetzten Flags sind die gleichen.

Prüfung auf spezifisches Bit (zur Laufzeit angegeben)

Dies wird normalerweise durch den folgenden C/C++ Code gelöst (verschiebe Wert um n Bits nach rechts und schneide dann niederwertigstes Bit ab):

Listing 1.280: C/C++

```
if ((value>>n)&1)
    ....
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

Listing 1.281: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Eine andere Möglichkeit: (verschiebe 1 Bit n-mal nach links, isoliere dieses Bit im Eingabewert und prüfe, ob es nicht 0 ist):

Listing 1.282: C/C++

```
if (value & (1<<n))
    ....
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

Listing 1.283: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

Setzen eines spezifischen Bits (zur Compilerzeit bekannt)

Listing 1.284: C/C++

```
value=value|0x40;
```

Listing 1.285: x86

```
OR REG, 40h
```

Listing 1.286: ARM (ARM Modus) and ARM64

```
ORR R0, R0, #0x40
```

Setzen eines spezifischen Bits (zur Laufzeit angegeben)

Listing 1.287: C/C++

```
value=value|(1<<n);
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

Listing 1.288: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

Löschen eines spezifischen Bits (zur Compilezeit bekannt)

Man verwendet einfach den AND Befehl mit dem invertierten Wert:

Listing 1.289: C/C++

```
value=value&(~0x40);
```

Listing 1.290: x86

```
AND REG, 0FFFFFFBFh
```

Listing 1.291: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

Dies sorgt dafür, dass alle Bits bis auf eines gesetzt werden.

ARM im ARM mode verfügt über den Befehl BIC, der wie ein NOT +AND Befehlspaar arbeitet:

Listing 1.292: ARM (ARM Modus)

```
BIC R0, R0, #0x40
```

Löschen eines spezifischen Bits (zur Laufzeit angegeben)

Listing 1.293: C/C++

```
value=value&(~(1<<n));
```

Listing 1.294: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

1.21.7 Übungen

- http://challenges.re/67
- http://challenges.re/68
- http://challenges.re/69
- http://challenges.re/70

1.22 Linearer Kongruenzgenerator als Pseudozufallszahlengenerator

Ein linearer Kongruenzgenerator ist die wohl einfachste Form der Zufallszahlenerzeugung.

Er wird heutzutage nicht mehr so häufig eingesetzt[139], kann aber, weil er so einfach ist (nur eine Multiplikation, eine Addition und eine AND-Operation), dennoch gut als Beispiel dienen.

[139] Der Mersenne-Twister ist besser

```
#include <stdint.h>

// Konstanten aus dem Numerical Recipes Buch
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}
```

Es gibt hier zwei Funktionen: die erste wird verwendet und den internen Zustand zu initialisieren und die zweite wird zum Erzeugen der Pseudozufallszahlen aufgerufen.

Wir sehen, dass im Algorithmus zwei Konstanten verwendet werden. Sie stamen aus [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)].

Definieren wir sie über den `#define` C/C++ Ausdruck. Es handelt sich um ein Makro.

Der Unterschied zwischen einem C/C++ Makro und einer Konstanten ist, dass alle Makros durch den C/C++ Präprozessor durch mit ihrem Wert ersetzt werden und dadurch im Gegensatz zu Variablen keinen Speicherplatz verbrauchen.

Eine Konstante ist im Gegensatz dazu eine nur lesbare Variable.

Es ist möglich einen Pointer (oder eine Adresse) einer Konstanten zu verwenden; das ist mit einem Makro nicht möglich.

Die letzte AND-Operation wird benötigt, da `my_rand()` gemäß C-Standard einen Wert zwischen 0 und 32767 zurückgeben muss.

Wenn man 32-Bit-Pseudozufallszahlen benötigt, kann die AND-Operation einfach weggelassen werden.

1.22.1 x86

Listing 1.295: Optimierender MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
```

```
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
_rand   ENDP

_TEXT   ENDS
```

Hier sehen wir, dass beide Konstanten in den Code eingebettet wurden. Es wurde kein Speicher für sie reserviert.

Die Funktion `my_srand()` kopiert ihren Eingabewert in die interne Variable `rand_state`.

`my_rand()` nimmt diese, berechnet den nächsten `rand_state`, schneidet ihn ab und belässt ihn im EAX Register.

Die nicht optimierte Version ist umfangreicher:

Listing 1.296: Nicht optimierender MSVC 2013

```
_BSS    SEGMENT
_rand_state DD  01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _init$[ebp]
        mov     DWORD PTR _rand_state, eax
        pop     ebp
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        push    ebp
        mov     ebp, esp
        imul    eax, DWORD PTR _rand_state, 1664525
        mov     DWORD PTR _rand_state, eax
        mov     ecx, DWORD PTR _rand_state
        add     ecx, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, ecx
        mov     eax, DWORD PTR _rand_state
        and     eax, 32767      ; 00007fffH
        pop     ebp
        ret     0
```

```
_rand   ENDP

_TEXT   ENDS
```

1.22.2 x64

Die x64 Version ist größtenteils identisch und verwendet 32-Bit-Register anstelle der 64-Bit-Register (wir arbeiten hier mit *int* Werten).

Die Funktion `my_srand()` nimmt seinen Eingabewert aus dem Register ECX und nicht vom Stack:

Listing 1.297: Optimierender MSVC 2013 x64

```
_BSS    SEGMENT
rand_state DD   01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = Eingabewert
        mov     DWORD PTR rand_state, ecx
        ret     0
my_srand ENDP

_TEXT   SEGMENT
my_rand PROC
        imul    eax, DWORD PTR rand_state, 1664525 ; 0019660dH
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

GCC erzeugt fast den gleichen Code.

1.22.3 32-bit ARM

Listing 1.298: Optimierender Keil 6/2013 (ARM Modus)

```
my_srand PROC
        LDR      r1,|L0.52|  ; lade Pointer auf rand_state
        STR      r0,[r1,#0]  ; speichere rand_state
        BX       lr
        ENDP

my_rand PROC
        LDR      r0,|L0.52|  ; lade Pointer auf rand_state
        LDR      r2,|L0.56|  ; lade RNG_a
        LDR      r1,[r0,#0]  ; lade rand_state
```

```
        MUL      r1,r2,r1
        LDR      r2,|L0.60|  ; lade RNG_c
        ADD      r1,r1,r2
        STR      r1,[r0,#0]  ; speichere rand_state
; AND mit 0x7FFF:
        LSL      r0,r1,#17
        LSR      r0,r0,#17
        BX       lr
        ENDP

|L0.52|
        DCD      ||.data||
|L0.56|
        DCD      0x0019660d
|L0.60|
        DCD      0x3c6ef35f

        AREA ||.data||, DATA, ALIGN=2

rand_state
        DCD      0x00000000
```

Es ist nicht möglich 32-Bit-Konstanten in ARM Befehle einzubetten, sodass Keil diese extern speichern und dann zusätzlich laden muss. Eine interessante Sache ist, dass es ebenfalls nicth möglich ist, die Konstante 0x7FFF einzubetten. Was Keil dann tut, ist *rand_state* um 17 Bits nach links zu verschieben und dann um 17 Bits nach rechts zu Verchieben. Die entspricht Ausdruck $(rand_state \ll 17) \gg 17$ in C/C++. Es scheint eine nutzlose Operation zu sein, löscht aber die oberen 17 Bits und lässt die 15 niederen Bits intakt und das ist genau was wir wollen.

Optimierender Keil für Thumb mode erzeugt fast den gleichen Code.

1.22.4 MIPS

Listing 1.299: Optimierender GCC 4.4.5 (IDA)

```
my_srand:
; speichere $a0 in rand_state:
                lui     $v0, (rand_state >> 16)
                jr      $ra
                sw      $a0, rand_state
my_rand:
; lade rand_state nach $v0:
                lui     $v1, (rand_state >> 16)
                lw      $v0, rand_state
                or      $at, $zero  ; lade delay slot
; multipliziere rand_state in $v0 mit 1664525 (RNG_a):
                sll     $a1, $v0, 2
                sll     $a0, $v0, 4
                addu    $a0, $a1, $a0
```

```
                sll     $a1, $a0, 6
                subu    $a0, $a1, $a0
                addu    $a0, $v0
                sll     $a1, $a0, 5
                addu    $a0, $a1
                sll     $a0, 3
                addu    $v0, $a0, $v0
                sll     $a0, $v0, 2
                addu    $v0, $a0
; addiere 1013904223 (RNG_c)
; der LI Befehl ist von IDA aus LUI und ORI zusammengesetzt
                li      $a0, 0x3C6EF35F
                addu    $v0, $a0
; speichere in rand_state:
                sw      $v0, (rand_state & 0xFFFF)($v1)
                jr      $ra
                andi    $v0, 0x7FFF ; branch delay slot
```

Hier sehen wir nur eine Konstante (0x3C6EF35F oder 1013904223). Wo befindet sich die andere (1664525)?

Es scheint, dass die Multiplikation mit 1664525 nur durch Verschieben und Addieren durchgeführt wird. Überprüfen wir diese Vermutung:

```
#define RNG_a 1664525

int f (int a)
{
        return a*RNG_a;
}
```

Listing 1.300: Optimierender GCC 4.4.5 (IDA)

```
f:
                sll     $v1, $a0, 2
                sll     $v0, $a0, 4
                addu    $v0, $v1, $v0
                sll     $v1, $v0, 6
                subu    $v0, $v1, $v0
                addu    $v0, $a0
                sll     $v1, $v0, 5
                addu    $v0, $v1
                sll     $v0, 3
                addu    $a0, $v0, $a0
                sll     $v0, $a0, 2
                jr      $ra
                addu    $v0, $a0, $v0 ; branch delay slot
```

Tatsächlich!

MIPS Relocation

Wir werden uns auch anschauen wie solche Operationen wie das Laden und Werten aus dem Speicher und das Speichern tatsächlich funktionieren.

Die Listings wurden mit IDA erzeugt, was einige Details versteckt.

Wir lassen objdump zweimal laufen: um das disassemblierte Listing und die Relocation List zu erhalten:

Listing 1.301: Optimierender GCC 4.4.5 (objdump)

```
# objdump -D rand_O3.o

...

00000000 <my_srand>:
   0:   3c020000        lui     v0,0x0
   4:   03e00008        jr      ra
   8:   ac440000        sw      a0,0(v0)

0000000c <my_rand>:
   c:   3c030000        lui     v1,0x0
  10:   8c620000        lw      v0,0(v1)
  14:   00200825        move    at,at
  18:   00022880        sll     a1,v0,0x2
  1c:   00022100        sll     a0,v0,0x4
  20:   00a42021        addu    a0,a1,a0
  24:   00042980        sll     a1,a0,0x6
  28:   00a42023        subu    a0,a1,a0
  2c:   00822021        addu    a0,a0,v0
  30:   00042940        sll     a1,a0,0x5
  34:   00852021        addu    a0,a0,a1
  38:   000420c0        sll     a0,a0,0x3
  3c:   00821021        addu    v0,a0,v0
  40:   00022080        sll     a0,v0,0x2
  44:   00441021        addu    v0,v0,a0
  48:   3c043c6e        lui     a0,0x3c6e
  4c:   3484f35f        ori     a0,a0,0xf35f
  50:   00441021        addu    v0,v0,a0
  54:   ac620000        sw      v0,0(v1)
  58:   03e00008        jr      ra
  5c:   30427fff        andi    v0,v0,0x7fff

...

# objdump -r rand_O3.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000000 R_MIPS_HI16       .bss
00000008 R_MIPS_LO16       .bss
0000000c R_MIPS_HI16       .bss
```

```
00000010 R_MIPS_LO16       .bss
00000054 R_MIPS_LO16       .bss

...
```

Betrachten wir die beiden Relocations für die Funktion `my_srand()`.

Die erste für die Adresse 0 hat den Typ R_MIPS_HI16. Die zweite für die Adresse 8 hat den Typ R_MIPS_LO16.

Das bedeutet, dass die Adresse zu Beginn des .bss Segments zu den Befehlen an der Adresse 0 (höherer Teil der Adresse) bzw. 8 (niederer Teil der Adresse) geschrieben wird.

Die Variable `rand_state` befindet sich ganz am Anfang des .bss Segments.

Wir finden hier Nullen in den Operanden der Befehle `LUI` und `SW`, da sich hier noch nichts befindet—der Compiler weiß noch nicht was hier hingeschrieben werden soll.

Der Linker wird dieses Problem beheben und der höhere Teil der Adresse wird in den Operanden von `LUI` geschriebne und der niedere Teil der Adresse in den Operanden von `SW`.

`SW` addiert den niederen Teil der Adresse und den Inhalt des Registers $V0 (hier befindet sich der höhere Teil).

Das gleiche geschieht mit der Funktion `my_rand()`: Die R_MIPS_HI16 Relocation teilt dem Linker mit, dass der höhere Teil der Adresse des .bss Segments in den Befehl `LUI` geschrieben wird.

Der höhere Teil der Adresse der Variablen rand_state befindet sich im Register $V1.

Der Befehl `LW` an der Adresse 0x10 addiert den höheren und niederen Teil und lädt den Wert der Variablen rand_state nach $V0.

Der Befehl `SW` an der Adresse 0x54 summiert erneut auf und speichert dann den neuen Wert in der globale Variable rand_state.

IDA arbeitet die Relocations beim Laden ab, sodass diese Details verborgen bleiben, aber wir sollten wissen, dass es sie gibt.

1.22.5 Threadsichere Version des Beispiels

Eine threadsichere Version des Beispiels wird später hier gezeigt:6.2.1 on page 592.

1.23 Strukturen

Ein C/C++ struct ist lediglich eine Menge von Variablen (nicht unbedingt gleichen Typs), die zusammen gespeichert werden. [140].

[140] AKA „heterogener Container“

1.23.1 MSVC: SYSTEMTIME Beispiel

Betrachten wir das SYSTEMTIME[141] struct in win32, das die Systemzeit beschreibt.

Das struct ist folgendermaßen definiert:

Listing 1.302: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Schreiben wir eine C-Funktion, um die aktuelle Zeit auszugeben:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

Wir erhalten das Folgende (MSVC 2010):

Listing 1.303: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
```

[141] MSDN: SYSTEMTIME structure

```
    push   eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

Für dieses struct werden 16 Byte im lokalen Stack reserviert —das entspricht genau `sizeof(WORD)*8` (es gibt 8 WORD Variablen in diesem struct).

Man beachte, dass dieses struct mit dem `wYear` Feld beginnt. Man kann sagen, dass ein Pointer auf das SYSTEMTIME struct an die Funktion `GetSystemTime()`[142] übergeben wird, aber man könnte auch sagen, dass ein Pointer auf das Feld `wYear` übergeben wird, denn dabei handelt es sich um dasselbe! `GetSystemTime()` schreibt das aktuelle Jahr in den WORD Pointer, verschiebt um 2 Byte, schreibt den aktuellen Monat, usw. usf.

[142] MSDN: SYSTEMTIME structure

OllyDbg

Kompilieren wir dieses Beispiel in MSVC 2010 mit `/GS- /MD` und laden es in OllyDbg.

Öffnen wir die Fenster für Daten und Stack an der Adresse, die als erstes Argument der Funktion `GetSystemTime()` übergeben wird und warten, bis das Programm an dieser Stelle ist. Wir sehen das folgende:

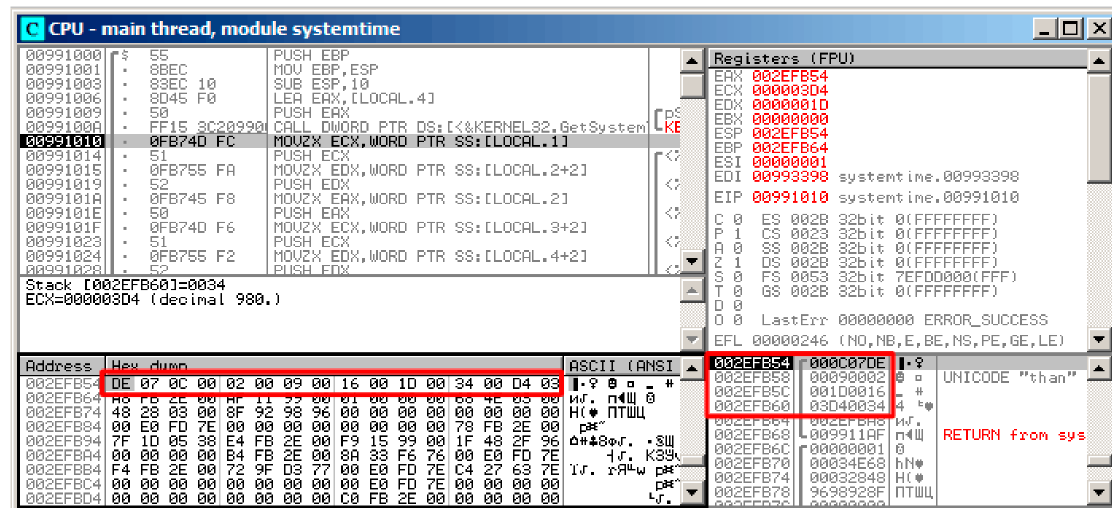


Abbildung 1.104: OllyDbg: `GetSystemTime()` wurde gerade ausgeführt

Die Systemzeit, die diese Ausführung der Funktion auf meinem Computer liefert, ist 9. Dezember 2014, 22:29:52:

Listing 1.304: `printf()` output

```
2014-12-09 22:29:52
```

Wir sehen also diese 16 Byte im Datenfenster:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

Je zwei Byte repräsentieren ein Feld des structs. Da die Endianess hier *little Endian* ist, finden wir das niederwertige Byte zuerst und danach das höherwertige.

Es werden also die folgenden Werte aktuell im Speicher gehalten:

Hexadezimalzahl	Dezimalzahl	Feldname
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

Wir finden die gleichen Werte im Stackfenster, aber sie werden als 32-Bit-Werte gruppiert.

Die Funktion `printf()` nimmt sich die Werte, die sie braucht und schreibt sie in die Konsole.

Manche Werte werden von `printf()` nicht ausgegeben (`wDayOfWeek` und `wMilliseconds`), aber sie sind im Speicher jederzeit für uns verfügbar.

Ein struct durch ein Array ersetzen

Die Tatsache, dass die Felder eines structs Variablen sind, die nebeneinander angeordnet sind, kann leicht durch folgendes Beispiel belegt werden. Wir erinnern uns an die Beschreibung des SYSTEMTIME structs und schreiben unser Beispiel wie folgt um:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond ↙
    ↘ */);

    return;
};
```

Der Compiler meckert ein wenig:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from '↙
    ↘ WORD [8]' to 'LPSYSTEMTIME'
```

Trotzdem erzeugt er den folgenden Code:

Listing 1.305: Nicht optimierender MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

```
_array$ = -16   ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

Dieses Programm funktioniert genau wie das erste!

Sehr interessant ist, dass dieser Assemblercode nicht vom entsprechenden Code des ersten Beispiels unterschieden werden kann.

Beim bloßen Ansehen des Codes kann man also nicht feststellen, ob ein struct oder ein Array deklariert wurde.

Trotzdem würde man letzteres nicht annehmen, da es sehr ungebräuchlich ist.

DIe Felder des structs können durch den Entwickler ausgetauscht oder verändert werden, etc.

Wir untersuchen dieses Beispiel nicht in OllyDbg, da es mit dem Beispiel mit dem struct identisch ist.

1.23.2 Reservieren von Platz für ein struct mit malloc()

Manchmal ist es einfacher structs nicht im lokalen Stack, sondern im Heap abzulegen:

```
#include <windows.h>
#include <stdio.h>

void main()
```

```
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Kompilieren wir das Beispiel nun mit Optimierung (/Ox), sodass leicht zu erkennen ist, was wir brauchen.

Listing 1.306: Optimierender MSVC

```
_main       PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main       ENDP
```

Also ist sizeof(SYSTEMTIME) = 16 und das entspricht exakt der Anzahl an Bytes, die von malloc() reserviert wurde. Die Funktion gibt einen Pointer auf den neu re-

servierten Speicherblock in das Register EAX zurück, welcher dann nach ESI verschoben wird. Die win32-Funktion `GetSystemTime()` kümmert sich um das Speichern des Wertes in ESI und das ist der Grund, warum er nicht hier gesichert wird und nach dem Aufruf von `GetSystemTime()` weiterverwendet wird.

Wir finden einen neuen Befehl —MOVZX (*Move with Zero eXtend*). Er wird in den meisten Fällen als MOVSX verwendet, setzt aber die übrigen Bits auf 0. Das wird benötigt, da `printf()` einen 32-Bit *int* erwartet, wir aber ein WORD im struct haben —also einen 16-Bit Typ ohne Vorzeichen. Das ist der Grund dafür, dass beim Kopieren eines Wertes aus einem WORD in einen *int* die Bits 16 bis 31 gelöscht werden müsse: hier könnten sich Zufallswerte vom Ergebnis einer vorherigen Operation auf diesem Register befinden.

In diesem Beispiel ist es möglich das struct als Array von 8 WORDs darzustellen:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};
```

Wir erhalten:

Listing 1.307: Optimierender MSVC

```
$SG78594 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
```

```
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Wieder erhalten wir Code, der nicht vom vorhergehenden zu unterscheiden ist.

Ebenfalls halten wir fest, dass man dies in der Praxis nicht macht, außer man weiß genau, was man tut.

1.23.3 UNIX: struct tm

Linux

Nehmen wir das struct `tm` aus `time.h` in Linux als Beispiel:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

Kompilieren wir das Beispiel mit GCC 4.4.1:

Listing 1.308: GCC 4.4.1

```
main proc near
     push    ebp
```

```
     mov     ebp, esp
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; erstes Argument für time()
     call    time
     mov     [esp+3Ch], eax
     lea     eax, [esp+3Ch]  ; nimm Pointer auf Rückgabewert von time()
     lea     edx, [esp+10h]  ; bei ESP+10h beginnt das struct tm
     mov     [esp+4], edx    ; übergib pointer auf den Beginn des structs
     mov     [esp], eax      ; übegib pointer auf Ergebnis von time()
     call    localtime_r
     mov     eax, [esp+24h]  ; tm_year
     lea     edx, [eax+76Ch] ; edx=eax+1900
     mov     eax, offset format ; "Year: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+20h]      ; tm_mon
     mov     eax, offset aMonthD ; "Month: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+1Ch]     ; tm_mday
     mov     eax, offset aDayD  ; "Day: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+18h]     ; tm_hour
     mov     eax, offset aHourD ; "Hour: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+14h]        ; tm_min
     mov     eax, offset aMinutesD ; "Minutes: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+10h]
     mov     eax, offset aSecondsD ; "Seconds: %d\n"
     mov     [esp+4], edx          ; tm_sec
     mov     [esp], eax
     call    printf
     leave
     retn
main endp
```

Aus irgendwelchen Gründen hat IDA hier nicht die Namen der lokalen Variablen im lokalen Stack hinzugeschrieben. Da wir aber bereits erfahrene Reverse Engineers sind, können wir in diesem einfachen Beispiel auch ohne diese Information auskommen.

Man beachte besonders den Befehl`lea edx, [eax+76Ch]` —dieser Befehl addiert lediglich `0x76C` (1900) zum Wert in EAX, ohne jedoch die Flags zu verändern. Siehe hierzu auch den entsprechenden Abschnitt über LEA (**??** on page ??).

GDB

Versuchen wir dieses Beispiel in GDB zu laden [143]:

Listing 1.309: GDB

```
dennis@ubuntuvm:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvm:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols ↙
    ↘ found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29      printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc:     0x080484c3      0x080485c0      0x000007de      0x00000000
0xbffff0ec:     0x08048301      0x538c93ed      0x00000025      0x0000000a
0xbffff0fc:     0x00000012      0x00000002      0x00000005      0x00000072
0xbffff10c:     0x00000001      0x00000098      0x00000001      0x00002a30
0xbffff11c:     0x0804b090      0x08048530      0x00000000      0x00000000
(gdb)
```

Wir finden unser struct leicht im Stack. Zuerst schauen wir uns an, wie es in *time.h* definiert ist:

Listing 1.310: time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

Man beachte, dass hier 32-Bit *int* anstelle von WORD in SYSTEMTIME verwendet werden, sodass jedes Feld 32 Bit belegt.

Hier sind die Felder unseres structs im Stack:

```
0xbffff0dc: 0x080484c3      0x080485c0      0x000007de       0x00000000
```

[143] Das Ergebnis von *date* ist zu Demonstrationszwecken leicht korrigiert worden. Natürlich ist es nicht möglich, GDB noch in der gleichen Sekunde auszuführen.

```
0xbffff0ec: 0x08048301      0x538c93ed      0x00000025 sec   0x0000000a min
0xbffff0fc: 0x00000012 hour 0x00000002 mday 0x00000005 mon   0x00000072 ↙
    ↳ year
0xbffff10c: 0x00000001 wday 0x00000098 yday 0x00000001 isdst 0x00002a30
0xbffff11c: 0x0804b090      0x08048530      0x00000000       0x00000000
```

Oder als Tabelle:

Hexadezimalzahl	Dezimalzahl	Feldname
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

Genau wie bei SYSTEMTIME(1.23.1 on page 405) sind auch hier noch weitere Felder verfügbar, die nicht verwendet werden, wie z.B. tm_wday, tm_yday, tm_isdst.

ARM

Optimierender Keil 6/2013 (Thumb Modus)

Das gleiche Beispiel:

Listing 1.311: Optimierender Keil 6/2013 (Thumb Modus)

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

       PUSH    {LR}
       MOVS    R0, #0          ; timer
       SUB     SP, SP, #0x34
       BL      time
       STR     R0, [SP,#0x38+timer]
       MOV     R1, SP          ; tp
       ADD     R0, SP, #0x38+timer ; timer
       BL      localtime_r
       LDR     R1, =0x76C
       LDR     R0, [SP,#0x38+var_24]
       ADDS    R1, R0, R1
       ADR     R0, aYearD      ; "Year: %d\n"
       BL      __2printf
       LDR     R1, [SP,#0x38+var_28]
```

```
        ADR     R0, aMonthD     ; "Month: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_2C]
        ADR     R0, aDayD       ; "Day: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_30]
        ADR     R0, aHourD      ; "Hour: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_34]
        ADR     R0, aMinutesD   ; "Minutes: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_38]
        ADR     R0, aSecondsD   ; "Seconds: %d\n"
        BL      __2printf
        ADD     SP, SP, #0x34
        POP     {PC}
```

Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

IDA erkennt das `tm` struct, da IDA die Typen der Argumente von Funktionen aus Bibliotheken wie `localtime_r()` kennt, sodass IDA hier die Zugriffe auf die Elemente des structs und deren Namen anzeigt.

Listing 1.312: Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
var_38 = -0x38
var_34 = -0x34

        PUSH {R7,LR}
        MOV  R7, SP
        SUB  SP, SP, #0x30
        MOVS R0, #0  ; time_t *
        BLX  _time
        ADD  R1, SP, #0x38+var_34 ; struct tm *
        STR  R0, [SP,#0x38+var_38]
        MOV  R0, SP  ; time_t *
        BLX  _localtime_r
        LDR  R1, [SP,#0x38+var_34.tm_year]
        MOV  R0, 0xF44 ; "Year: %d\n"
        ADD  R0, PC  ; char *
        ADDW R1, R1, #0x76C
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
```

```
       LDR  R1, [SP,#0x38+var_34.tm_min]
       MOV  R0, 0xF28 ; "Minutes: %d\n"
       ADD  R0, PC  ; char *
       BLX  _printf
       LDR  R1, [SP,#0x38+var_34]
       MOV  R0, 0xF25 ; "Seconds: %d\n"
       ADD  R0, PC  ; char *
       BLX  _printf
       ADD  SP, SP, #0x30
       POP  {R7,PC}

...

00000000 tm        struc ; (sizeof=0x2C, standard type)
00000000 tm_sec    DCD ?
00000004 tm_min    DCD ?
00000008 tm_hour   DCD ?
0000000C tm_mday   DCD ?
00000010 tm_mon    DCD ?
00000014 tm_year   DCD ?
00000018 tm_wday   DCD ?
0000001C tm_yday   DCD ?
00000020 tm_isdst  DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone   DCD ? ; offset
0000002C tm        ends
```

MIPS

Listing 1.313: Optimierender GCC 4.4.5 (IDA)

```
main:

; IDA erkennt die Feldnamen im struct nicht, wir haben sie manuell benannt:

var_40          = -0x40
var_38          = -0x38
seconds         = -0x34
minutes         = -0x30
hour            = -0x2C
day             = -0x28
month           = -0x24
year            = -0x20
var_4           = -4

  lui     $gp, (__gnu_local_gp >> 16)
  addiu   $sp, -0x50
  la      $gp, (__gnu_local_gp & 0xFFFF)
  sw      $ra, 0x50+var_4($sp)
  sw      $gp, 0x50+var_40($sp)
  lw      $t9, (time & 0xFFFF)($gp)
  or      $at, $zero ; load delay slot, NOP
  jalr    $t9
```

```
  move    $a0, $zero ; branch delay slot, NOP
  lw      $gp, 0x50+var_40($sp)
  addiu   $a0, $sp, 0x50+var_38
  lw      $t9, (localtime_r & 0xFFFF)($gp)
  addiu   $a1, $sp, 0x50+seconds
  jalr    $t9
  sw      $v0, 0x50+var_38($sp) ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+year($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  la      $a0, $LC0        # "Year: %d\n"
  jalr    $t9
  addiu   $a1, 1900 ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+month($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  lui     $a0, ($LC1 >> 16)  # "Month: %d\n"
  jalr    $t9
  la      $a0, ($LC1 & 0xFFFF)  # "Month: %d\n" ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+day($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  lui     $a0, ($LC2 >> 16)  # "Day: %d\n"
  jalr    $t9
  la      $a0, ($LC2 & 0xFFFF)  # "Day: %d\n" ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+hour($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  lui     $a0, ($LC3 >> 16)  # "Hour: %d\n"
  jalr    $t9
  la      $a0, ($LC3 & 0xFFFF)  # "Hour: %d\n" ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+minutes($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  lui     $a0, ($LC4 >> 16)  # "Minutes: %d\n"
  jalr    $t9
  la      $a0, ($LC4 & 0xFFFF)  # "Minutes: %d\n" ; branch delay slot
  lw      $gp, 0x50+var_40($sp)
  lw      $a1, 0x50+seconds($sp)
  lw      $t9, (printf & 0xFFFF)($gp)
  lui     $a0, ($LC5 >> 16)  # "Seconds: %d\n"
  jalr    $t9
  la      $a0, ($LC5 & 0xFFFF)  # "Seconds: %d\n" ; branch delay slot
  lw      $ra, 0x50+var_4($sp)
  or      $at, $zero ; load delay slot, NOP
  jr      $ra
  addiu   $sp, 0x50

$LC0:           .ascii "Year: %d\n"<0>
$LC1:           .ascii "Month: %d\n"<0>
$LC2:           .ascii "Day: %d\n"<0>
$LC3:           .ascii "Hour: %d\n"<0>
$LC4:           .ascii "Minutes: %d\n"<0>
```

```
76 $LC5:          .ascii "Seconds: %d\n"<0>
```

Dieses hier ist ein Beispiel, in dem die Branch Delay Slots uns verwirren können.

Es gibt zum Beispiel den Befehl `addiu $a1, 1900` in Zeile 35, der 1900 zur Jahreszahl hinzuaddiert.

Wir dürfen nicht vergessen, dass er vor dem zugehörigen JALR in Zeile 34 ausgeführt wird.

Struct als Menge von Werten

Um zu veranschaulichen, dass ein struct nur eine Menge von nebeneinanderliegenden Variablen ist, überarbeiten wir unser Beispiel, indem wir auf die Definition des *tm* structs schauen:Listing.1.310.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday↙
    ↳ , tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

Der Pointer auf das Feld `tm_sec` wird nach `localtime_r` übergeben, d.h. an das erste Element des structs.

Der Compiler warnt uns:

Listing 1.314: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from ↙
    ↳ incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of ↙
    ↳ type 'int *'
```

Trotzdem erzeugt er folgenden Code:

Listing 1.315: GCC 4.7.3

```
main      proc near

var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
tm_year   = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 30h
          call    __main
          mov     [esp+30h+var_30], 0 ; arg 0
          call    time
          mov     [esp+30h+unix_time], eax
          lea     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          lea     eax, [esp+30h+unix_time]
          mov     [esp+30h+var_30], eax
          call    localtime_r
          mov     eax, [esp+30h+tm_year]
          add     eax, 1900
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mon]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mday]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_hour]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_min]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
          call    printf
          leave
          retn
```

```
main        endp
```

Dieser Code ist zum vorherigen identisch und es ist unmöglich zu sagen, ob es sich im originalen Quellcode um ein struct oder nur um eine Menge von Variablen handelt.

Es funktioniert also, ist aber in der Praxis nicht empfehlenswert.

Nicht optimierende Compiler legen normalerweise Variablen auf dem lokalen Stack in der Reihenfolge an, in der sie in der Funktion deklariert wurden.

Ein Garantie dafür gibt es freilich nicht.

Andere Compiler könnten an dieser Stelle übrigens davor warnen, dass die Variablen `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` - nicht aber `tm_sec` - ohne Initialisierung verwendet werden.

Der Compiler weiß nicht, dass diese durch die Funktion `localtime_r()` befüllt werden.

Wir haben dieses Beispiel ausgewählt, da alle Felder im struct vom Typ *int* sind.

Es würde nicht funktionieren, wenn die Felder 16 Bit (`WORD`) groß wären, wie im Beispiel des `SYSTEMTIME` structs—`GetSystemTime()` würde sie falsch befüllen (da die lokalen Variablen auf 32-Bit-Grenzen angeordnet sind). Mehr dazu im folgenden Abschnitt: „Felder in Strukturen packen“ (1.23.4 on page 425).

Ein struct ist also nichts als eine Menge von an einer Stelle gespeicherten Variablen. Man kan sagen, dass das struct ein Befehl an den Compiler ist, diese Variablen an einer Stelle zu halten. In ganz frühen Versionen von C (vor 1972) gab es übrigens gar keine structs [Dennis M. Ritchie, *The development of the C language*, (1993)][144].

Dieses Beispiel wird nicht im Debugger gezeigt, da es dem gerade gezeigten entspricht.

Struct als Array aus 32-Bit-Worten

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
```

[144] Auch verfügbar als pdf

```
    };
};
```

Wir können einen Pointer auf ein struct in ein Array aus *int*s casten und es funktioniert. Wir lassen dieses Beispiel zur Systemzeit 23:51:45 26-July-2014 laufen.

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)
```

Die Variablen sind hier in der gleichen Reihenfolge, in der die in der Definition des structs aufgezählt werden:1.310 on page 414.

Hier ist der erzeugte Code:

Listing 1.316: Optimierender GCC 4.8.1

```
main proc near
     push    ebp
     mov     ebp, esp
     push    esi
     push    ebx
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; timer
     lea     ebx, [esp+14h]
     call    _time
     lea     esi, [esp+38h]
     mov     [esp+4], ebx    ; tp
     mov     [esp+10h], eax
     lea     eax, [esp+10h]
     mov     [esp], eax      ; timer
     call    _localtime_r
     nop
     lea     esi, [esi+0]       ; NOP
loc_80483D8:
; EBX ist hier ein Pointer auf das struct,
; ESI ist der Pointer auf dessen Ende.
     mov     eax, [ebx] ; hole 32-Bit-Wort aus dem Array
     add     ebx, 4          ; nächstes Feld im struct
     mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
     mov     dword ptr [esp], 1
     mov     [esp+0Ch], eax     ; übergib Wert an printf()
     mov     [esp+8], eax       ; übergib Wert an printf()
     call    ___printf_chk
     cmp     ebx, esi        ; Ende des structs?
     jnz     short loc_80483D8  ; nein - dann lade nächsten Wert
     lea     esp, [ebp-8]
```

```
     pop     ebx
     pop     esi
     pop     ebp
     retn
main endp
```

Tatsächlich: der Platz auf dem lokalen Stack wird zuerst wie in struct und dann wie ein Array behandelt.

Es ist sogar möglich, die Felder des structs über diesen Pointer zu verändern.

Und wiederum ist es zweifellos ein seltsamer Weg die Dinge umzusetzen; er ist für produktiven Code definitiv nicht empfehlenswert.

Übung

Versuchen Sie als Übung die Monatsnummer zu verändern (um 1 zu erhöhen), indem Sie das struct wie ein Array behandeln.

Struct als Bytearray

Wir können sogar noch weiter gehen. Casten wir den Pointer zu einem Bytearray und ziehen einen Dump:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    };
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
```

```
0x01 0x00 0x00 0x00
```

Wir haben dieses Beispiel auch zur Systemzeit 23:51:45 26-July-2014 ausgeführt [145]. Die Werte sind genau dieselben wie im vorherigen Dump(1.23.3 on page 422) und natürlich steht das LSB vorne, da es sich um eine Little-Endian-Architektur handelt(**??** on page ??).

Listing 1.317: Optimierender GCC 4.8.1

```
main proc near
     push    ebp
     mov     ebp, esp
     push    edi
     push    esi
     push    ebx
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; timer
     lea     esi, [esp+14h]
     call    _time
     lea     edi, [esp+38h]  ; Ende des structs
     mov     [esp+4], esi    ; tp
     mov     [esp+10h], eax
     lea     eax, [esp+10h]
     mov     [esp], eax      ; timer
     call    _localtime_r
     lea     esi, [esi+0]       ; NOP
; ESI ist hier der Pointer auf das struct im lokalen Stack.
; EDI ist der Pointer auf das Ende des structs
loc_8048408:
     xor     ebx, ebx        ; j=0

loc_804840A:
     movzx   eax, byte ptr [esi+ebx]    ; lade Byte
     add     ebx, 1          ; j=j+1
     mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
     mov     dword ptr [esp], 1
     mov     [esp+8], eax       ; übergib geladenes Byte an printf()
     call    ___printf_chk
     cmp     ebx, 4
     jnz     short loc_804840A
; gib carriage return (Wagenrücklauf) Zeichen (CR) aus
     mov     dword ptr [esp], 0Ah ; c
     add     esi, 4
     call    _putchar
     cmp     esi, edi        ; Ende des structs?
     jnz     short loc_8048408 ; j=0
     lea     esp, [ebp-0Ch]
     pop     ebx
     pop     esi
     pop     edi
     pop     ebp
```

[145] Datum und Uhrzeit sind zu Demonstrationszwecken identisch. Die Bytewerte sind modifiziert.

```
    retn
main endp
```

1.23.4 Felder in Strukturen packen

Ein wichtiges Thema ist das Packen von Feldern in structs.

Betrachten wir ein einfaches Beispiel:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

Wie wir sehen haben wir zwei *char* Felder (jedes ist exakt ein Byte groß) und zwei weitere vom Typ *int* (zu je 4 Byte).

x86

Das Beispiel kompiliert zu folgendem Code:

Listing 1.318: MSVC 2012 /GS- /Ob0

```
_tmp$ = -16
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    mov    BYTE PTR _tmp$[ebp], 1     ; setze Feld a
    mov    DWORD PTR _tmp$[ebp+4], 2  ; setze Feld b
    mov    BYTE PTR _tmp$[ebp+8], 3   ; setze Feld c
    mov    DWORD PTR _tmp$[ebp+12], 4 ; setze Feld d
```

```
10      sub    esp, 16                ; reserviere Platz für temporäres struct
11      mov    eax, esp
12      mov    ecx, DWORD PTR _tmp$[ebp]  ; kopiere unser struct in das temporäre
13      mov    DWORD PTR [eax], ecx
14      mov    edx, DWORD PTR _tmp$[ebp+4]
15      mov    DWORD PTR [eax+4], edx
16      mov    ecx, DWORD PTR _tmp$[ebp+8]
17      mov    DWORD PTR [eax+8], ecx
18      mov    edx, DWORD PTR _tmp$[ebp+12]
19      mov    DWORD PTR [eax+12], edx
20      call    _f
21      add    esp, 16
22      xor    eax, eax
23      mov    esp, ebp
24      pop    ebp
25      ret    0
26  _main    ENDP
27
28  _s$ = 8 ; size = 16
29  ?f@@YAXUs@@@Z PROC ; f
30      push   ebp
31      mov    ebp, esp
32      mov    eax, DWORD PTR _s$[ebp+12]
33      push   eax
34      movsx  ecx, BYTE PTR _s$[ebp+8]
35      push   ecx
36      mov    edx, DWORD PTR _s$[ebp+4]
37      push   edx
38      movsx  eax, BYTE PTR _s$[ebp]
39      push   eax
40      push   OFFSET $SG3842
41      call   _printf
42      add    esp, 20
43      pop    ebp
44      ret    0
45  ?f@@YAXUs@@@Z ENDP ; f
46  _TEXT    ENDS
```

Wir übergeben das struct als Ganzes, aber im Code können wir sehen, dass das struct in ein temporäres struct kopiert wird (ein Platz hierfür wird in Zeile 10 auf dem Stack reserviert), und dass dann alle 4 Felder einzeln (in den Zeilen 12…19) kopiert werden und anschließend ihr Pointer (Adresse) übergeben wird.

Das struct wird kopiert, da nicht bekannt ist, ob die Funktion `f()` das struct verändern wird oder nicht. Wenn es verändert wird, muss das struct in `main()` auf dem vorherigen Stand bleiben.

Wir könnten C/C++ Pointer verwenden und der erzeugte Code wäre fast der gleiche nur ohne das Kopieren.

Wie wir sehen können, wird die Adresse von jedem Feld auf einer 4 Byte Grenze angeordnet. Das ist der Grund dafür, dass jeder *char* hier 4 Byte belegt (wie ein *int*). Es ist für die CPU einfacher auf Speicher an entsprechend angeordneten Adressen

zuzugreifen und Daten von dort im Cache zwischenzuspeichern.

Trotzdem ist dieses Vorgehen nicht besonders ökonomisch.

Komplieren wir mit der Option (/Zp1)(*/Zp[n] pack structures on n-byte boundary*).

Listing 1.319: MSVC 2012 /GS- /Zp1

```
_main    PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 12
    mov    BYTE PTR _tmp$[ebp], 1     ; setze Feld a
    mov    DWORD PTR _tmp$[ebp+1], 2  ; setze Feld b
    mov    BYTE PTR _tmp$[ebp+5], 3   ; setze Feld c
    mov    DWORD PTR _tmp$[ebp+6], 4  ; setze Feld d
    sub    esp, 12                    ; reserviere Platz für temporäres
    struct
    mov    eax, esp
    mov    ecx, DWORD PTR _tmp$[ebp]  ; kopiere 10 Byte
    mov    DWORD PTR [eax], ecx
    mov    edx, DWORD PTR _tmp$[ebp+4]
    mov    DWORD PTR [eax+4], edx
    mov    cx, WORD PTR _tmp$[ebp+8]
    mov    WORD PTR [eax+8], cx
    call    _f
    add    esp, 12
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_TEXT    SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC    ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+6]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+5]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+1]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP    ; f
```

Das struct benötigt nun lediglich 10 Byte und jeder *char* Wert genau 1 Byte. Welchen Vorteil bringt uns das? In erster Linie spart man Platz. Ein Nachteil hieran —die CPU

greift hier auf die Felder langsamer zu als es möglich wäre.

Das struct wird auch in `main()` kopiert. Nicht Feld für Feld, sondern alle 10 Byte direkt durch Verwendung von drei Paaren von `MOV` Befehlen. Warum aber nicht 4 Paare?

Der Compiler hat entschieden, dass es besser ist, die 10 Byte mit 3 `MOV` Befehlspaaren zu kopieren als zwei 32-Bit-Worte und dann zwei Byte mit insgesamt 4 `MOV` Paaren.

Solch eine Implementierung, die zum Kopieren `MOV` anstelle eines Aufrufs von `memcpy()` verwendet, ist sehr gebräuchlich, da es schneller ist als ein Funktionsaufruf von `memcpy()`—zumindest für kurze Blöcke: **??** on page ??. Man kann sich leicht überlegen, dass, wenn ein struct in vielen Quellcode und Objektdateien verwendet wird, alle diese mit der gleichen Konvention bezüglich des Packens in structs kompiliert werden müssen.

Neben der Option Zp in MSVC, die die Anordnung der Felder von structs festlegt, gibt es auch die Compileroption `#pragma pack`, die direkt im Quellcode definiert werden kann. Sie ist sowohl in MSVC[146]als auch in GCC[147] verfügbar.

Gehen wir zurück zum SYSTEMTIME struct, das aus 16-Bit-Feldern besteht. Wie kann unser Compiler wissen, dass diese in 1-Byte-Anordnung gepackt werden müssen?

Die Datei `WinNT.h` enthält dies:

Listing 1.320: WinNT.h

```
#include "pshpack1.h"
```

Und dies:

Listing 1.321: WinNT.h

```
#include "pshpack4.h"                    // 4 byte packing is the default
```

Die Datei PshPack1.h sieht wie folgt aus:

Listing 1.322: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

Dies sagt dem Compiler wie die structs, die hinter `#pragma pack` definiert werden, gepackt werden müssen.

[146] MSDN: Working with Packing Structures
[147] Structure-Packing Pragmas

OllyDbg + standardmäßig gepackte Felder

Betrachten wir unser Beispiel (in dem die Felder standardmäßig auf 4 Byte angeordnet werden) in OllyDbg:

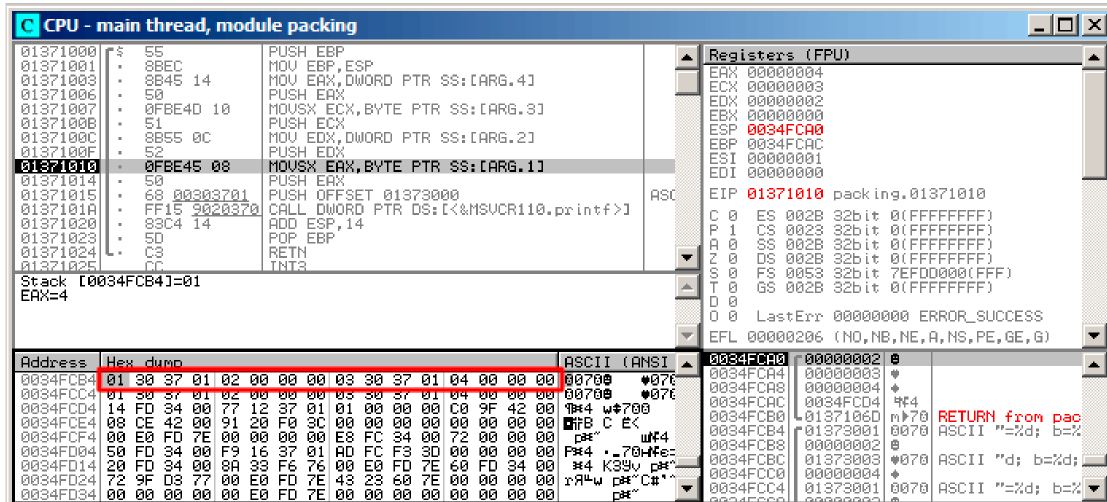


Abbildung 1.105: OllyDbg: vor der Ausführung von `printf()`

Wir sehen unsere 4 Felder im Datenfenster.

Wir fragen uns aber, woher die Zufallsbytes (0x30, 0x37, 0x01) stammen, die neben dem ersten (a) und dritten (c) Feld liegen.

Betrachten wir unser Listing 1.318 on page 425, erkennen wir, dass das erste und dritte Feld vom Typ *char* ist, und daher nur ein Byte geschrieben wird, nämlich 1 bzw. 3 (Zeilen 6 und 8).

Die übrigen 3 Byte des 32-Bit-Wortes werden im Speicher nicht verändert! Deshalb befinden sich hier zufällige Reste.

Diese Reste beeinflussen den Output von `printf()` in keinster Weise, da die Werte für die Funktion mithilfe von MOVSX vorbereitet werden, der Bytes und nicht Worte als Argumente hat: Listing.1.318 (Zeilen 34 und 38). Der vorzeichenerweiternde Befehl MOVSX wird hier übrigens verwendet, da *char* standardmäßig in MSVC und GCC vorzeichenbehaftet ist. Würde hier der Datentyp `unsigned char` oder `uint8_t` verwendet, würde der Befehl MOVZX stattdessen verwendet.

OllyDbg + Felder auf 1 Byte Grenzen angeordnet

Hier sind die Dinge viel klarer ersichtlich: 4 Felder benötigen 16 Byte und die Werte werden nebeneinander gespeichert.

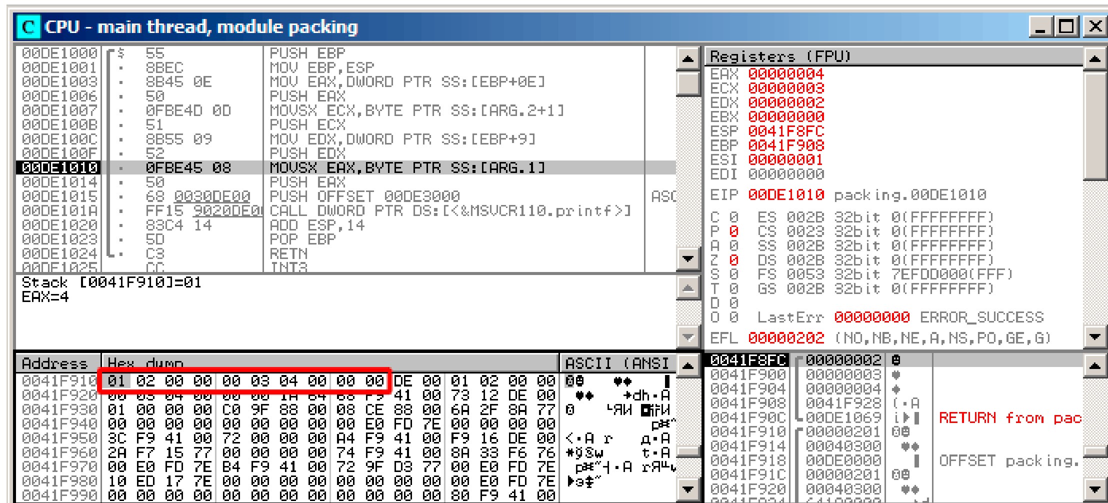


Abbildung 1.106: OllyDbg: Vor der Ausführung von `printf()`

ARM

Optimierender Keil 6/2013 (Thumb Modus)

Listing 1.323: Optimierender Keil 6/2013 (Thumb Modus)

```
.text:0000003E              exit ; CODE XREF: f+16
.text:0000003E 05 B0             ADD     SP, SP, #0x14
.text:00000040 00 BD             POP     {PC}


.text:00000280              f
.text:00000280
.text:00000280              var_18 = -0x18
.text:00000280              a      = -0x14
.text:00000280              b      = -0x10
.text:00000280              c      = -0xC
.text:00000280              d      = -8
.text:00000280
.text:00000280 0F B5             PUSH    {R0-R3,LR}
.text:00000282 81 B0             SUB     SP, SP, #4
.text:00000284 04 98             LDR     R0, [SP,#16]    ; d
.text:00000286 02 9A             LDR     R2, [SP,#8]     ; b
.text:00000288 00 90             STR     R0, [SP]
.text:0000028A 68 46             MOV     R0, SP
.text:0000028C 03 7B             LDRB    R3, [R0,#12]    ; c
.text:0000028E 01 79             LDRB    R1, [R0,#4]     ; a
```

```
.text:00000290 59 A0            ADR     R0, aADBDCDDD   ; "a=%d; b=%d;
    c=%d; d=%d\n"
.text:00000292 05 F0 AD FF      BL      __2printf
.text:00000296 D2 E6            B       exit
```

Wir sehen, dass hier ein struct anstelle eines Pointers auf ein struct übergeben wird und da die ersten vier Funktionsargumente in ARM über Register übergeben werden, werden die Felder des structs mittels R0-R3 übergeben.

LDRB lädt ein Byte aus dem Speicher und erweitert es unter Berücksichtigung des Vorzeichens auf 32 Bit. Dies entspricht MOVSX in x86. Hier wird der Befehl verwendet, um die Felder a und a des structs zu laden.

Eine weitere Sache, die ins Auge sticht, ist, dass anstelle eines Funktionsepilogs ein Sprung zum Epilog einer anderen Funktion vorliegt. Bei der anderen handelt es sich um eine komplett unterschiedliche Funktion, die in keiner Weise mit unserer zusammenhängt, aber denselben Epilog hat (möglicherweise, da diese ebenfalls 5 lokale Variablen enthält ($5 * 4 = 0x14$)).

Außerdem befindet sie sich im Code in der Nähe (man vergleiche die Adressen).

Tatsächlich spielt es auch keine Rolle, welcher Epilog ausgeführt wird, solange dieser wie gewünscht funktioniert.

Offensichtlich hat Keil aus ökonomischen Gründen entschieden, einen Teil der anderen Funktion wiederzuverwenden.

Der Epilog benötigt 4 Bytes—während der Sprung nur 2 verbraucht.

ARM + Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

Listing 1.324: Optimierender Xcode 4.6.3 (LLVM) (Thumb-2 Modus)

```
var_C = -0xC

      PUSH   {R7,LR}
      MOV    R7, SP
      SUB    SP, SP, #4
      MOV    R9, R1 ; b
      MOV    R1, R0 ; a
      MOVW   R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
      SXTB   R1, R1  ; prepare a
      MOVT.W R0, #0
      STR    R3, [SP,#0xC+var_C] ; place d to stack for printf()
      ADD    R0, PC  ; format-string
      SXTB   R3, R2  ; prepare c
      MOV    R2, R9  ; b
      BLX    _printf
      ADD    SP, SP, #4
      POP    {R7,PC}
```

SXTB (*Signed Extend Byte*) ist analog zu MOVSX in x86. Der ganze Rest ist identisch.

MIPS

Listing 1.325: Optimierender GCC 4.4.5 (IDA)

```
f:

var_18          = -0x18
var_10          = -0x10
var_4           = -4
arg_0           =  0
arg_4           =  4
arg_8           =  8
arg_C           =  0xC

; $a0=s.a
; $a1=s.b
; $a2=s.c
; $a3=s.d
                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x28
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x28+var_4($sp)
                sw      $gp, 0x28+var_10($sp)
; prepare a byte from 32-bit big-endian integer:
                sra     $t0, $a0, 24
                move    $v1, $a1
; prepare a byte from 32-bit big-endian integer:
                sra     $v0, $a2, 24
                lw      $t9, (printf & 0xFFFF)($gp)
                sw      $a0, 0x28+arg_0($sp)
                lui     $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d\n"
                sw      $a3, 0x28+var_18($sp)
                sw      $a1, 0x28+arg_4($sp)
                sw      $a2, 0x28+arg_8($sp)
                sw      $a3, 0x28+arg_C($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "a=%d; b=%d; c=%d; d=%d\n"
                move    $a1, $t0
                move    $a2, $v1
                jalr    $t9
                move    $a3, $v0 ; branch delay slot
                lw      $ra, 0x28+var_4($sp)
                or      $at, $zero ; load delay slot, NOP
                jr      $ra
                addiu   $sp, 0x28 ; branch delay slot

$LC0:           .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>
```

Die Felder des structs landen in den Registern $A0..$A3 und werden dann für `printf()` nach $A1..$A3 verschoben, während das vierte Feld (aus $A3) mit SW über den lokalen Stack übergeben wird.

Wir fragen uns aber, warum es hier zwei SRA („Shift Word Right Arithmetic“) Befehle für die beiden *char* Felder gibt.

MIPS ist eine Big-Endian-Architektur**??** on page ?? und das Debian Linux, mit dem wir arbeiten, ist ebenfalls Big Endian.

Wenn nun Byte Variablen in 32-Bit-Feldern im struct gespeichert werden, besetzen sie die hohen Bits 24..31.

Wenn ein Variable vom Typ *char* zu einem 32-Bit-Wert erweitert werden muss, muss sie um 24 Bits nach rechts verschoben werden.

char ist ein vorzeichenbehafteter Typ, sodass hier eine arithmetische Verschiebung anstelle einer logischen erfolgen muss.

Eine Sache noch

Ein struct als Funktionsargument zu übergeben (anstelle eines Pointers auf ein struct) ist das gleiche wie alle Felder des structs einzeln zu übergeben.

Wenn die Felder im struct standardmäßig gepackt werden, kann die Funktion f() wie folgt neu geschrieben werden:

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

Das führt schlussendlich zum gleichen Code.

1.23.5 Verschachtelte structs

Wir betrachten nun Situationen, in denen ein struct innerhalb eines anderen definiert ist.

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
```

```
};

int main()
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

Beide Felder von `inner_struct` werden hier zwischen den Feldern a,b und d,e von `outer_struct` platziert.

Kompilieren wir mit (MSVC 2010):

Listing 1.326: Optimierender MSVC 2010 /Ob0

```
$SG2802 DB    'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT    SEGMENT
_s$ = 8
_f    PROC
    mov    eax, DWORD PTR _s$[esp+16]
    movsx  ecx, BYTE PTR _s$[esp+12]
    mov    edx, DWORD PTR _s$[esp+8]
    push   eax
    mov    eax, DWORD PTR _s$[esp+8]
    push   ecx
    mov    ecx, DWORD PTR _s$[esp+8]
    push   edx
    movsx  edx, BYTE PTR _s$[esp+8]
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call   _printf
    add    esp, 28
    ret    0
_f    ENDP

_s$ = -24
_main    PROC
    sub    esp, 24
    push   ebx
    push   esi
    push   edi
    mov    ecx, 2
    sub    esp, 24
    mov    eax, esp
; from this moment, EAX is synonymous to ESP:
```

```
    mov     BYTE PTR _s$[esp+60], 1
    mov     ebx, DWORD PTR _s$[esp+60]
    mov     DWORD PTR [eax], ebx
    mov     DWORD PTR [eax+4], ecx
    lea     edx, DWORD PTR [ecx+98]
    lea     esi, DWORD PTR [ecx+99]
    lea     edi, DWORD PTR [ecx+2]
    mov     DWORD PTR [eax+8], edx
    mov     BYTE PTR _s$[esp+76], 3
    mov     ecx, DWORD PTR _s$[esp+76]
    mov     DWORD PTR [eax+12], esi
    mov     DWORD PTR [eax+16], ecx
    mov     DWORD PTR [eax+20], edi
    call    _f
    add     esp, 24
    pop     edi
    pop     esi
    xor     eax, eax
    pop     ebx
    add     esp, 24
    ret     0
_main    ENDP
```

Eine Kuriosität ist hier, dass wir, wenn wir in den Assemblercode schauen, nicht einmal feststellen können, dass hier ein struct innerhalb eines anderen verwendet wurde! Durch diese Beobachtung können wir sagen, dass verschachtelte structs in lineare oder eindimensionale structs ausgefaltet werden.

Wenn wir aber die Deklaration von `struct inner_struct c;` durch `struct inner_struct *c;` ersetzen (also einen Pointer verwenden), sieht die Sache ganz anders aus.

OllyDbg

Laden wir dieses Beispiel in OllyDbg und schauen uns `outer_struct` im Speicher an:

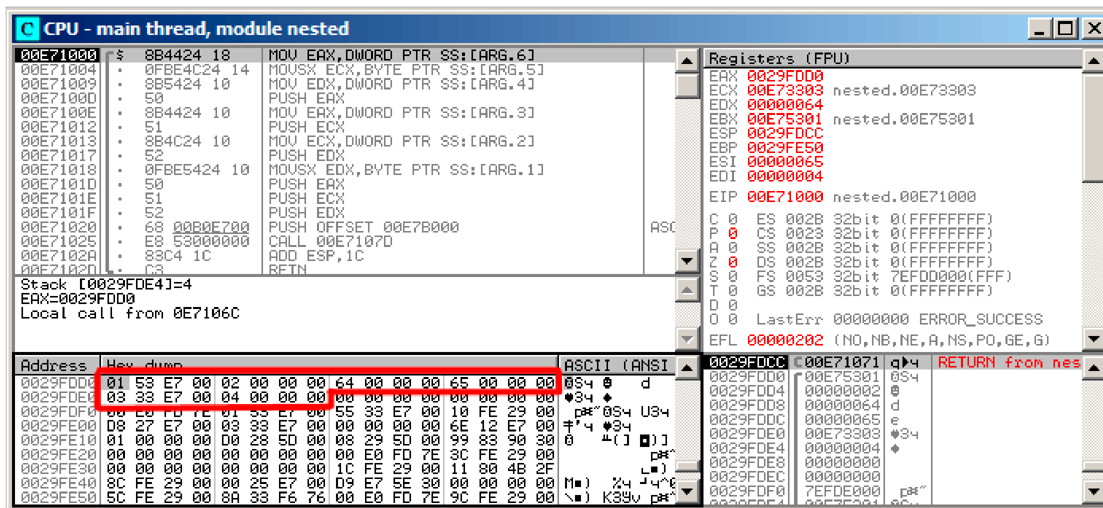


Abbildung 1.107: OllyDbg: vor der Ausführung von `printf()`

Die Werte werden wie folgt im Speicher abgelegt:

- *(outer_struct.a)* (byte) 1 + 3 Byte mit zufälligen Werten;
- *(outer_struct.b)* (32-bit word) 2;
- *(inner_struct.a)* (32-bit word) 0x64 (100);
- *(inner_struct.b)* (32-bit word) 0x65 (101);
- *(outer_struct.d)* (byte) 3 + 3 Byte mit zufälligen Werten;
- *(outer_struct.e)* (32-bit word) 4.

1.23.6 Bitfields in einem struct

CPUID Beispiel

Die Sprache C/C++ erlaubt die Definition der exakten Anzahl von Bits für jedes Feld in einem struct. Das ist sehr nützlich, wenn man Speicherplatz sparen muss. Zum Beispiel genügt ein Bit für eine *bool* Variable. Natürlich ist dieses Vorgehen nicht angebracht, wenn die Geschwindigkeit wichtig ist.

Betrachten wir das Beispiel des `CPUID`[148]Befehls. Dieser Befehl liefert Informationen über die aktuelle CPU und ihre Eigenschaften.

[148] wikipedia

Wenn EAX vor der Ausführung des Befehls auf 1 gesetzt ist, liefert CPUID diese Informationen gepackt in das EAX Register zurück:

3:0 (4 bits)	Schrittweite
7:4 (4 bits)	Modell
11:8 (4 bits)	Familie
13:12 (2 bits)	Prozessortype
19:16 (4 bits)	Erweitertes Modell
27:20 (8 bits)	Erweiterte Familie

MSVC 2010 verfügt über ein CPUID Makro, aber GCC 4.4.1 nicht. Erstellen wir also für uns eine solche Funktion in GCC, indem wir den built-in Assembler[149] verwenden.

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];
```

[149] Mehr zum internen GCC Assembler

```
    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

Nachdem CPUID die Register EAX/EBX/ECX/EDX befüllt hat, werden deren Inhalte in das Array b[] geschrieben. Danach haben wir einen Pointer auf das CPUID_1_EAX struct und zeigen auf den Wert in EAX aus dem Array b[].

Mit anderen Worten: wir behandeln einen 32-Bit *int* wie ein struct. Danach lesen wir spezifische Bits aus dem struct.

MSVC

Kompilieren wir das Beispiel in MSVC 2008 mit der Option /Ox:

Listing 1.327: Optimierender MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
```

```
    and    edx, 15
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 16
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call   _printf

    shr    esi, 20
    and    esi, 255
    push   esi
    push   OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call   _printf
    add    esp, 48
    pop    esi

    xor    eax, eax
    pop    ebx

    add    esp, 16
    ret    0
_main    ENDP
```

Der Befehl SHR verschiebt den Wert in EAX um die Anzahl der Bits die überprungen werden müssen, d.h. wir ignorieren einige Bits am rechten Rand.

Der Befehl AND löscht die nicht benötigten Bits am linken Rand bzw. belässt nur die Bits in EAX, die wir auch benötigen.

MSVC + OllyDbg

Laden wir unser Beispiel in OllyDbg und schauen welche Werte sich nach der Ausführung von CPUID in den Register EAX/EBX/ECX und EDX befinden:

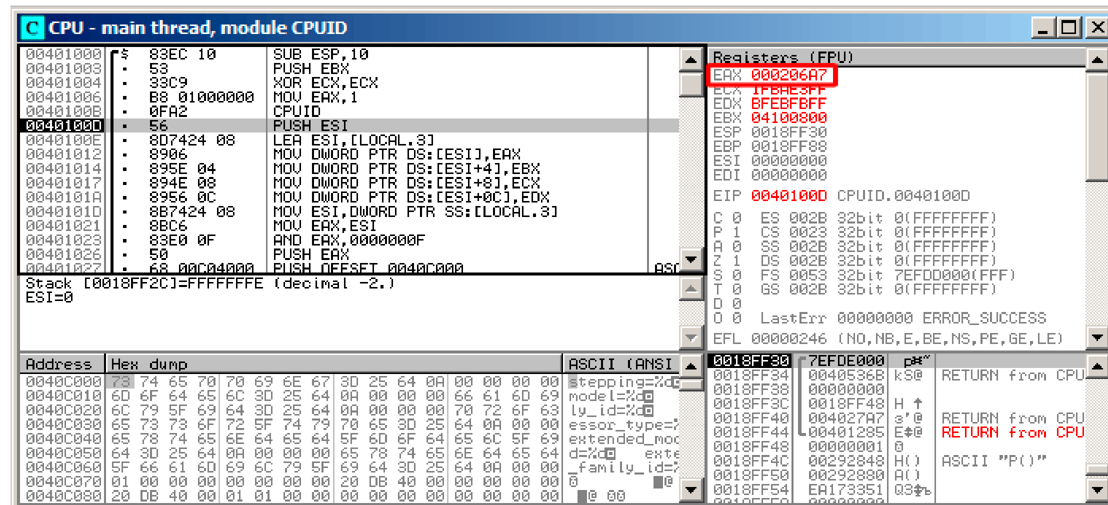


Abbildung 1.108: OllyDbg: Nach Ausführung von CPUID

EAX enthält `0x000206A7` (meine CPU ist ein Intel Xeon E3-1220).
Dies entspricht $0b00000000000000100000011010100111$ in Binärdarstellung.

Die Bits werden wie folgt durch die Felder aufgeteilt:

Feld	in binär	in dezimal
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Listing 1.328: Console output

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

GCC

Versuchen wir es mit GCC 4.4.1 mit der Option -O3

Listing 1.329: Optimierender GCC 4.4.1

```
main            proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ;
    "extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     [esp+8], esi
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
```

```
    call    ___printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

Fast das gleiche. Das einzig Bemerkenswerte ist, dass GCC die Berechnung von `extended_model_id` und `extended_family_id` in einem Block kombiniert, anstatt sie vor jedem Aufruf von `printf()` getrennt zu berechnen.

Gleitkomma Datentypen als Struktur behandeln

Wie wir bereits im Abschnitt über die FPU (1.19 on page 253) besprochen haben, bestehen *float* und *double* jeweils aus einem Vorzeichen, einem Signifikanden (oder Bruch) und einem Exponenten. Wir wollen am Beispiel des Typs *float* untersuchen, ob wir direkt mit diesen Feldern arbeiten können.



(S — Vorzeichen)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // Bruch
    unsigned int exponent : 8;  // Exponent + 0x3FF
    unsigned int sign : 1;      // Vorzeichenbit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multipliziere d mit 2^n(n ist hier 2)

    memcpy (&f, &t, sizeof (float));
```

```
    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

Das struct `float_as_struct` belegt den gleichen Speicherplatz wie ein *float*, d.h., 4 Byte oder 32 Bit.

Wir setzen das negative Vorzeichen im Eingabewert und multiplizieren die gesamte Zahl mit 2^2, d.h. 4, indem wir zum Exponenten 2 hinzuaddieren.

Kompilieren wir das Beispiel mit MSVC 2008 ohne Optimierung:

Listing 1.330: Nicht optimierender MSVC 2008

```
_t$ = -8   ; size = 4
_f$ = -4   ; size = 4
__in$ = 8  ; size = 4
?f@@YAMM@Z PROC ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - setze Minuszeichen
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - entferne Signifikanden
    and    eax, 255         ; 000000ffH - lasse nur Exponenten hier
    add    eax, 2           ; addiere 2
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - verschiebe Ergebnis auf Bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - entferne Exponenten

; addiere Originalwert ohne Exponenten und neu berechneten Exponenten:
    or     ecx, eax
    mov    DWORD PTR _t$[ebp], ecx
```

```
    push   4
    lea    edx, DWORD PTR _t$[ebp]
    push   edx
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    call   _memcpy
    add    esp, 12

    fld    DWORD PTR _f$[ebp]

    mov    esp, ebp
    pop    ebp
    ret    0
?f@@YAMM@Z ENDP   ; f
```

Ein wenig redundant. Hätten wir mit dem Flag /Ox kompiliert, wäre dort ein Aufruf von TTmemcpy(), sondern die Variable f wäre direkt verwendet worden. Einfacher ist es aber auf jeden Fall, sich die nicht optimierte Version anzuschauen.

Was würde GCC 4.4.1 mit -O3 hier tun?

Listing 1.331: Optimierender GCC 4.4.1

```
; f(float)
       public _Z1ff
_Z1ff  proc near

var_4  = dword ptr -4
arg_0  = dword ptr  8

       push    ebp
       mov     ebp, esp
       sub     esp, 4
       mov     eax, [ebp+arg_0]
       or      eax, 80000000h  ; setze Minuszeichen
       mov     edx, eax
       and     eax, 807FFFFFh  ; lasse nur Vorzeichen und Signifikanden in
    EAX
       shr     edx, 23         ; bereite Exponenten vor
       add     edx, 2          ; addiere 2
       movzx   edx, dl         ; lösche alle Bits außer 7:0 in EDX
       shl     edx, 23         ; verschiebe berechneten Exponenten
       or      eax, edx        ; kombiniere neuen Exponenten und Originalwert
    ohne Exponenten
       mov     [ebp+var_4], eax
       fld     [ebp+var_4]
       leave
       retn
_Z1ff  endp

       public main
main   proc near
       push    ebp
       mov     ebp, esp
       and     esp, 0FFFFFFF0h
```

```
        sub     esp, 10h
        fld     ds:dword_8048614 ; -4.936
        fstp    qword ptr [esp+8]
        mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
        mov     dword ptr [esp], 1
        call    ___printf_chk
        xor     eax, eax
        leave
        retn
main    endp
```

Die Funktion `f()` ist fast verständlich. Viel interessanter ist jedoch, dass GCC trotz des Sammelsuriums an Feldern in der Lage war, das Ergebnis von `f(1.234)` während des Kompilierens zu berechnen und der Funktion `printf()` während der Compilezeit als vorberechneten Wert bereitzustellen.

1.23.7 Übungen

- http://challenges.re/71
- http://challenges.re/72

1.24 Unions

Die
Cpp *union* wird hauptsächlich verwendet um eine Variable (oder einen Speicherblock) eines Datentyps als Variable eines anderen Datentyps zu interpretieren.

1.24.1 Pseudozufallszahlengenerator Beispiel

Wenn wir Zufallszahlen vom Typ *float* zwischen 0 und 1 brauchen, ist die einfachste Möglichkeit einen PRNG wie den Mersenne-Twister zu verwenden. Er produziert vorzeichenlose 32-Bit-Werte (mit anderen Worten: er erzeugt 32 zufällige Bits). Diesen Wert können wir in einen *float* umwandeln und dann durch `RAND_MAX` (`0xFFFFFFFF` in unserem Fall) teilen—wir erhalten einen Wert im Intervall 0..1.

Wie wir jedoch wissen, ist die Division langsam. Auch würden wir gerne so wenig FPU Operation wie möglich verwenden. Deshalb fragen wir uns, ob wir die Division loswerden können.

Erinnern wir uns an den Aufbau einer Fließkommazahl: sie besteht aus einem Vorzeichenbit, Bits im Signifikanden und Bits im Exponenten. Wir müssen also lediglich Zufallsbits in allen Bits des Signifikanden speichern um einen zufällige Fließkommazahl zu erhalten!

Der Exponent kann nicht null sein (in diesem Fall ist die Fließkommazahl denormalisiert); also speichern wir `0b01111111` im Exponenten—das bedeutet, dass der Exponent 1 ist. Dann füllen wir den Signifikanden mit Zufallsbits, setzen das Vorzeichenbit auf 0 (entspricht einer positiven Zahl) und voilà. Die erzeugte Zahl liegt zwischen 1 und 2; wir müssen also am Ende noch 1 abziehen.

Ein sehr einfacher linearer Kongruenzgenerator für Zufallszahlen wird in meinem Beispiel[150] vorgestellt: er erzeugt 32-Bit-Zahlen. Der PRNG wird mit der aktuellen Zeit um UNIX-Timestamp-Format initialisiert.

Wir stellen hier den Typ *float* als *union* dar–das ist die C/C++ Konstruktion, die es uns ermöglicht, einen Speicherblock als unterschiedliche Typen aufzufassen. In unserem Fall sind wir in der Lage eine Variable vom Typ *union* zu erzeugen und dann auf diese wie auf einen *float* oder *uint32_t* zuzugreifen. Man kann sagen, dass es sich dabei um einen Hack, d.h. Trick handelt. Sogar um einen sehr schmutzigen.

Der PRNG Code für Integer ist der bereits betrachtete: 1.22 on page 397. Deshalb verzichten wir an dieser Stelle auf ein erneutes Listing des kompilierten Codes.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// Integer PRNG Definitionen, Daten und Routinen:

// Konstanten aus dem Numerical Recipes Buch
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // globale Variable

void my_srand(uint32_t i)
{
        RNG_state=i;
};

uint32_t my_rand()
{
        RNG_state=RNG_state*RNG_a+RNG_c;
        return RNG_state;
};

// FPU PRNG Definitionen und Routinen:

union uint32_t_float
{
        uint32_t i;
        float f;
};

float float_rand()
{
        union uint32_t_float tmp;
        tmp.i=my_rand() & 0x007fffff | 0x3F800000;
        return tmp.f-1;
};

// test
```

[150] die Idee stammt von: URL

```
int main()
{
        my_srand(time(NULL)); // PRNG Initialisierung

        for (int i=0; i<100; i++)
                printf ("%f\n", float_rand());

        return 0;
};
```

x86

Listing 1.332: Optimierender MSVC 2010

```
$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=Pseudozufallswert
        and     eax, 8388607            ; 007fffffH
        or      eax, 1065353216         ; 3f800000H
; EAX=Pseudozufallswert & 0x007fffff | 0x3f800000
; speichere ihn auf lokalem Stack:
        mov     DWORD PTR _tmp$[esp+4], eax
; lade ihn erneut als float:
        fld     DWORD PTR _tmp$[esp+4]
; subtrahiere 1.0:
        fsub    QWORD PTR __real@3ff0000000000000
; speichere erhaltenen Wert auf dem lokalen Stack und lade ihn erneut:
; diese Befehle sind redundant:
        fstp    DWORD PTR tv130[esp+4]
        fld     DWORD PTR tv130[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP

_main   PROC
        push    esi
        xor     eax, eax
        call    _time
        push    eax
        call    ?my_srand@@YAXI@Z
        add     esp, 4
        mov     esi, 100
$LL3@main:
        call    ?float_rand@@YAMXZ
        sub     esp, 8
        fstp    QWORD PTR [esp]
```

```
        push    OFFSET $SG4238
        call    _printf
        add     esp, 12
        dec     esi
        jne     SHORT $LL3@main
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Die Namen der Funktionen sind hier so seltsam, weil das Beispiel als C++ kompiliert wurde, und hier name mangling in C++ vorliegt. Dies werden wir später besprechen: **??** on page ??. Wenn wir das Beispiel mit MSVC 2012 kompilieren, verwendet es SIMD Befehle für die FPU; mehr dazu unter: 1.29.5 on page 523.

ARM (ARM Modus)

Listing 1.333: Optimierender GCC 4.6.3 (IDA)

```
float_rand
                STMFD   SP!, {R3,LR}
                BL      my_rand
; R0=Pseudozufallswert
                FLDS    S0, =1.0
; S0=1.0
                BIC     R3, R0, #0xFF000000
                BIC     R3, R3, #0x800000
                ORR     R3, R3, #0x3F800000
; R3=Pseudorzufallswert & 0x007fffff | 0x3f800000
; kopiere aus R3 zur FPU (Register S15).
; entspricht bitweisem Kopieren, Umwandlung findet nicht statt:
                FMSR    S15, R3
; subtrahiere 1.0 und lasse Ergebnis in S0:
                FSUBS   S0, S15, S0
                LDMFD   SP!, {R3,PC}

flt_5C          DCFS 1.0

main
                STMFD   SP!, {R4,LR}
                MOV     R0, #0
                BL      time
                BL      my_srand
                MOV     R4, #0x64 ; 'd'

loc_78
                BL      float_rand
; S0=Pseudozufallswert
                LDR     R0, =aF         ; "%f"
; kovertiere float in double (printf() benötigt dieses Format):
                FCVTDS  D7, S0
; bitwises Kopieren von D7 nach R2/R3 (für printf()):
                FMRRD   R2, R3, D7
```

```
                BL      printf
                SUBS    R4, R4, #1
                BNE     loc_78
                MOV     R0, R4
                LDMFD   SP!, {R4,PC}

aF              DCB "%f",0xA,0
```

Wir ziehen auch einen Dump in objdump und shen, dass die FPU Befehle andere Namen als in IDA haben. Offenbar haben die Entwickler von IDA und binutils unterschiedliche Handbücher verwendet. Möglicherweise ist es hilfreich, beide Varianten der Befehlsnamen zu kennen.

Listing 1.334: Optimierender GCC 4.6.3 (objdump)

```
00000038 <float_rand>:
  38:   e92d4008        push    {r3, lr}
  3c:   ebfffffe        bl      10 <my_rand>
  40:   ed9f0a05        vldr    s0, [pc, #20]   ; 5c <float_rand+0x24>
  44:   e3c034ff        bic     r3, r0, #-16777216      ; 0xff000000
  48:   e3c33502        bic     r3, r3, #8388608        ; 0x800000
  4c:   e38335fe        orr     r3, r3, #1065353216     ; 0x3f800000
  50:   ee073a90        vmov    s15, r3
  54:   ee370ac0        vsub.f32        s0, s15, s0
  58:   e8bd8008        pop     {r3, pc}
  5c:   3f800000        svccc   0x00800000

00000000 <main>:
   0:   e92d4010        push    {r4, lr}
   4:   e3a00000        mov     r0, #0
   8:   ebfffffe        bl      0 <time>
   c:   ebfffffe        bl      0 <main>
  10:   e3a04064        mov     r4, #100        ; 0x64
  14:   ebfffffe        bl      38 <main+0x38>
  18:   e59f0018        ldr     r0, [pc, #24]   ; 38 <main+0x38>
  1c:   eeb77ac0        vcvt.f64.f32    d7, s0
  20:   ec532b17        vmov    r2, r3, d7
  24:   ebfffffe        bl      0 <printf>
  28:   e2544001        subs    r4, r4, #1
  2c:   1afffff8        bne     14 <main+0x14>
  30:   e1a00004        mov     r0, r4
  34:   e8bd8010        pop     {r4, pc}
  38:   00000000        andeq   r0, r0, r0
```

Die Befehle an den Stellen 0x5c in `float_rand()` und 0x38 in `main()` sind (Pseudo-)Zufallsrauschen.

1.24.2 Berechnung der Maschinengenauigkeit

Die Maschinengenauigkeit ist der kleinstmögliche Wert, mit dem die FPU arbeiten kann. Je mehr Bits für eine Fließkommazahl verwendet werden, desto kleiner ist die

Maschinengenauigkeit. Sie beträgt $2^{-23} \approx 1.19e-07$ für *float* und $2^{-52} \approx 2.22e-16$ für *double*. Siehe auch:Wikipedia article.

Interessant ist, wie einfach die Maschinengenauigkeit berechnet werden kann:

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
        uint32_t i;
        float f;
};

float calculate_machine_epsilon(float start)
{
        union uint_float v;
        v.f=start;
        v.i++;
        return v.f-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

Was wir hier machen ist, den Bruch in der IEEE 754 Zahl als Integer zu behandeln und 1 hinzuzuaddieren. Die resultierende Fließkommazahl ist gleich $starting_value + machine_epsilon$, sodass wir nur den Startwert (in der Fließkommaarithmetik) abziehen müssen um zu messen, welchen Unterschied ein Bit in der einfachen Genauigkeit (*float*) ausmacht. Die *union* dient hier als Mittel, um auf die IEEE 754 Zahl als regulären Integer zuzugreifen. Die Addition von 1 entspricht hier einer Addition von 1 zum Bruch in der Zahl, aber natürlich kann hier ein Overflow auftreten, was eine Addition von 1 zum Exponenten nach sich zieht.

x86

Listing 1.335: Optimierender MSVC 2010

```
tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
        fld     DWORD PTR _start$[esp-4]
; dieser Befehl ist redundant:
        fst     DWORD PTR _v$[esp-4]
        inc     DWORD PTR _v$[esp-4]
        fsubr   DWORD PTR _v$[esp-4]
; dieses Befehlspaar ist auch redundant:
        fstp    DWORD PTR tv130[esp-4]
        fld     DWORD PTR tv130[esp-4]
        ret     0
```

```
_calculate_machine_epsilon ENDP
```

Der zweite FST Befehl ist redundant: es besteht keine Notwendigkeit, den Eingabewert an derselben Stelle zu speichern (der Compiler hat entschieden, die Variable v an der gleichen Stelle im lokalen Stack anzulegen wie den Eingabewert). Der Wert wird mit INC um 1 erhöht als wäre es einen normale Integervariable. Danach wir der Wert als 32-Bit IEEE 754 Zahl in die FPU geladen, FSUBR erledigt den Rest und das Ergebnis wird in ST0 gespeichert. Das letzte FSTP/FLD Befehlspaar ist redundant, aber der Compiler hat es hier nicht wegoptimiert.

ARM64

Erweitern wir unser Beispiel auf 64 Bit:

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
        uint64_t i;
        double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
        uint_double v;
        v.d=start;
        v.i++;
        return v.d-start;
}

void main()
{
        printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 kennt keinen Befehl, der eine Zahl zu einem FPU D-Register addieren kann, sodass der Eingabewert (in D0) zunächst nach GPR kopiert wird, dann inkrementiert wird und schließlich in das FPU Register D1 kopiert wird, bevor die Subtraktion ausgeführt wird.

Listing 1.336: Optimierender GCC 4.9 ARM64

```
calculate_machine_epsilon:
        fmov    x0, d0       ; lade Eingabewert vom Typ double nach X0
        add     x0, x0, 1    ; X0++
        fmov    d1, x0       ; verschiebe ihn ins FPU Register
        fsub    d0, d1, d0   ; subtrahiere
        ret
```

Schauen Sie sich dieses Beispiel auch für x64 kompiliert mit SIMB Befehlen an:1.29.4 on page 522.

MIPS

Der neue Befehl ist hier MTC1 („Move To Coprocessor 1“): er überträgt Daten von GPR in die Register der FPU.

Listing 1.337: Optimierender GCC 4.4.5 (IDA)

```
calculate_machine_epsilon:
                mfc1    $v0, $f12
                or      $at, $zero ; NOP
                addiu   $v1, $v0, 1
                mtc1    $v1, $f2
                jr      $ra
                sub.s   $f0, $f2, $f12 ; branch delay slot
```

Fazit

Es ist schwer zu sagen, ob jemand eine solche Trickserei in echtem Produktivcode benötigt, aber wie bereits mehrfach erwähnt, ist dieses Beispiel gut geeignet, um das IEEE 754 Format und *unions* in C/C++ zu erklären.

1.24.3 FSCALE Ersatz

Agner Fog schreibt in seiner Abhandlung *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* [151] , dass der Befehl FSCALE FPU (der 2^n berechnet) auf vielen CPUs langsam ist und bietet einen schnelleren Ersatz an.

Hier ist meine Übersetzung von seinem Assemblercode in C/C++:

```
#include <stdint.h>
#include <stdio.h>

union uint_float
{
        uint32_t i;
        float f;
};

float flt_2n(int N)
{
        union uint_float tmp;

        tmp.i=(N<<23)+0x3f800000;
        return tmp.f;
};

struct float_as_struct
{
        unsigned int fraction : 23;
        unsigned int exponent : 8;
```

[151] http://www.agner.org/optimize/optimizing_assembly.pdf

```
        unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
        struct float_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x7f;
        return *(float*)(&tmp);
};

union uint64_double
{
        uint64_t i;
        double d;
};

double dbl_2n(int N)
{
        union uint64_double tmp;

        tmp.i=((uint64_t)N<<52)+0x3ff0000000000000UL;
        return tmp.d;
};

struct double_as_struct
{
        uint64_t fraction : 52;
        int exponent : 11;
        int sign : 1;
};

double dbl_2n_v2(int N)
{
        struct double_as_struct tmp;

        tmp.fraction=0;
        tmp.sign=0;
        tmp.exponent=N+0x3ff;
        return *(double*)(&tmp);
};

int main()
{
        // 2^11 = 2048
        printf ("%f\n", flt_2n(11));
        printf ("%f\n", flt_2n_v2(11));
        printf ("%lf\n", dbl_2n(11));
        printf ("%lf\n", dbl_2n_v2(11));
};
```

Der Befehl FSCALE kann zwar in bestimmten Umgebungen schneller sein, ist aber vor allem ein gutes Beispiel für *unions* und die Tatsache, dass der Exponent in der Form 2^n gespeichert wird, sodass ein Eingabewert n zum Exponenten nach IEEE 754 Standard verschoben wird. Der Exponent wird dann durch Addition von 0x3f800000 oder 0x3ff0000000000000 korrigiert.

Das gleiche kann ohne Verschiebung durch ein *struct* erreicht werden, aber intern werden stets Schiebebefehle verwendet.

1.24.4 Schnelle Berechnung der Quadratwurzel

Ein anderer bekannter Algorithmus, in dem *float* als *int* interpretiert wird, ist die schnelle Berechnung einer Quadratwurzel.

Listing 1.338: Quellcode stammt aus der Wikipedia: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots

```
/* Assumes that float is in the IEEE 754 single precision floating point ↙
    ↳ format
 * and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     *
     * ((((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((↙
    ↳ b + 1) / 2) * 2^m)
     *
     * where
     *
     * b = exponent bias
     * m = number of mantissa bits
     *
     * .
     */

    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */

    return *(float*)&val_int; /* Interpret again as float */
}
```

Versuchen Sie als Übung, diese Funktion zu kompilieren und zu verstehen wie sie funktioniert.

Es gibt auch einen bekannten Algorithmus zur schnellen Berechnung von $\frac{1}{\sqrt{x}}$. Der Algorithmus wurde vermutlich so populär, weil er in Quake III Arena verwendet wurde. Eine Beschreibung des Algorithmus' findet man bei Wikipedia: http://en.wikipedia.org/wiki/Fast_inverse_square_root.

1.25 Pointer auf Funktionen

Ein Pointer auf eine Funktion ist genau wie alle anderen Pointer nur die Adresse, an der die Funktion im Codesegment beginnt.

Sie werden oft für Callback-Funktionen verwendet.

Bekannte Beispiele hierfür sind:

- `qsort()`, `atexit()` aus der Standard C Bibliothek;
- *NIX OS Signale;
- Thread Start: `CreateThread()` (win32), `pthread_create()` (POSIX);
- viele win32 Funktionen wie `EnumChildWindows()`.
- viele Stellen im Linux kernel, zum Beispiel werden die Treiber für das Dateisystem über Callback-Funktionen aufgerufen.
- Die GCC Plugin-Functions werden ebenfalls über Callbacks aufgerufen.

Die Funktion `qsort()` ist eine Implementierung von Quicksort in der C/C++ Standardbibliothek. Die Funktion ist in der Lage jeden Datentyp zu sortieren, solange dieser über eine Funktion zum Vergleich von zwei Elementen verfügt und `qsort()` in der Lage ist, diese aufzurufen.

Die Vergleichsfunktion kann wie folgt definiert werden:

```
int (*compare)(const void *, const void *)
```

Verwenden wir das folgende Beispiel:

```
/* ex3 Sorting ints with qsort */

#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
  const int *a=(const int *)_a;
  const int *b=(const int *)_b;

  if (*a==*b)
    return 0;
  else
    if (*a < *b)
        return -1;
     else
      return 1;
}

int main(int argc, char* argv[])
{
   int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
   int i;

```

```
  /* Sort the array */
  qsort(numbers,10,sizeof(int),comp) ;
  for (i=0;i<9;i++)
    printf("Number = %d\n",numbers[ i ]) ;
  return 0;
}
```

1.25.1 MSVC

Kompilieren wir es in MSVC 2010 (einige Teile wurden zur Verkürzung weggelassen) mit der Option /Ox:

Listing 1.339: Optimierender MSVC 2010: /GS- /MD

```
__a$ = 8                                                ; size = 4
__b$ = 12                                               ; size = 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     ecx, DWORD PTR [ecx]
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp   ENDP


_numbers$ = -40                                         ; size = 40
_argc$ = 8                                              ; size = 4
_argv$ = 12                                             ; size = 4
_main   PROC
        sub     esp, 40                                 ; 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; 0000043fH
```

```
        mov     DWORD PTR _numbers$[esp+92], 88         ; 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; fffe7960H
        call    _qsort
        add     esp, 16                                 ; 00000010H

...
```

Bis hierhin nicht überraschend. Als viertes Argument wird die Adresse des Labels _comp übergeben, die eine Stelle ist, an der sich comp() befindet, oder, mit anderen Worten: die Adresse des ersten Befehls dieser Funktion.

Wie kann qsort() diese Funktion aufrufen?

Schauen wir uns diese Funktion, die sich in MSVCR80.DLL (ein MSVC DLL Modul mit Standard-C-Bibliotheksfunktionen) befindet, näher an:

Listing 1.340: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int
    (__cdecl *)(const void *, const void *))
.text:7816CBF0                  public _qsort
.text:7816CBF0 _qsort           proc near
.text:7816CBF0
.text:7816CBF0 lo               = dword ptr -104h
.text:7816CBF0 hi               = dword ptr -100h
.text:7816CBF0 var_FC           = dword ptr -0FCh
.text:7816CBF0 stkptr           = dword ptr -0F8h
.text:7816CBF0 lostk            = dword ptr -0F4h
.text:7816CBF0 histk            = dword ptr -7Ch
.text:7816CBF0 base             = dword ptr  4
.text:7816CBF0 num              = dword ptr  8
.text:7816CBF0 width            = dword ptr  0Ch
.text:7816CBF0 comp             = dword ptr  10h
.text:7816CBF0
.text:7816CBF0                  sub     esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                          ; CODE XREF: _qsort+B1
.text:7816CCE0                  shr     eax, 1
.text:7816CCE2                  imul    eax, ebp
.text:7816CCE5                  add     eax, ebx
.text:7816CCE7                  mov     edi, eax
.text:7816CCE9                  push    edi
.text:7816CCEA                  push    ebx
.text:7816CCEB                  call    [esp+118h+comp]
.text:7816CCF2                  add     esp, 8
.text:7816CCF5                  test    eax, eax
.text:7816CCF7                  jle     short loc_7816CD04
```

comp—ist das vierte Funktionsargument. Hier wird der Control Flow an die Adresse in comp Argument übergeben. Davor werden zwei Argumente für comp() vorbereitet. Das Ergebnis wird nach der Ausführung überprüft.

Das ist ein Grund, warum es gefährlich ist, Pointer auf Funktionen zu verwenden. Erstens: Wenn man `qsort()` mit einem falschen Funktionspointer aufruft, könnte `qsort()` den Control Flow an eine falsche Stelle übergeben, der Prozess könnte abstürzen und dieser Bug wäre sehr schwer zu finden.

Zweitens: Die Typen der Callback-Funktion müssen ganz genau passend sein, denn der Aufruf der falschen Funktion mit falschen Argumenten oder falschen Typen kann zu ernsthaften Problemen führen. Das Abstürzen des Prozesses ist hier kein Problem —das Problem ist die Ursache für den Absturz zu finden —, da der Compiler während des Kompilierens die potentiellen Probleme nicht entdeckt hat.

MSVC + OllyDbg

Laden wir unser Beispiel in OllyDbg und setzen einen Breakpoint auf `comp()`. Wir sehen wie dir Werte beim ersten Aufruf von `comp()` verglichen werden:

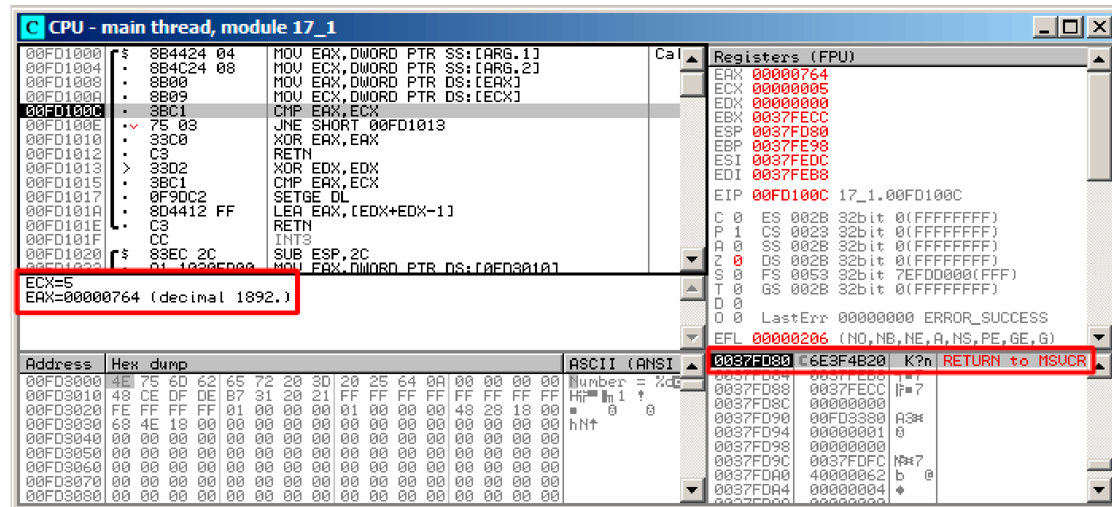


Abbildung 1.109: OllyDbg: erster Aufruf von `comp()`

OllyDbg zeigt die verglichenen Werte im Fenster unter dem Codefenster an. Wir können auch erkennen, dass der **SP!** auf RA zeigt, wo sich die Funktion `qsort()` (innerhalb von `MSVCR100.DLL`) befindet.

Durch Drücken von (F8) bis zum Befehl RETN und einmaliges weiteres Drücken von F8 kehren wir zur Funktion `qsort()` zurück:

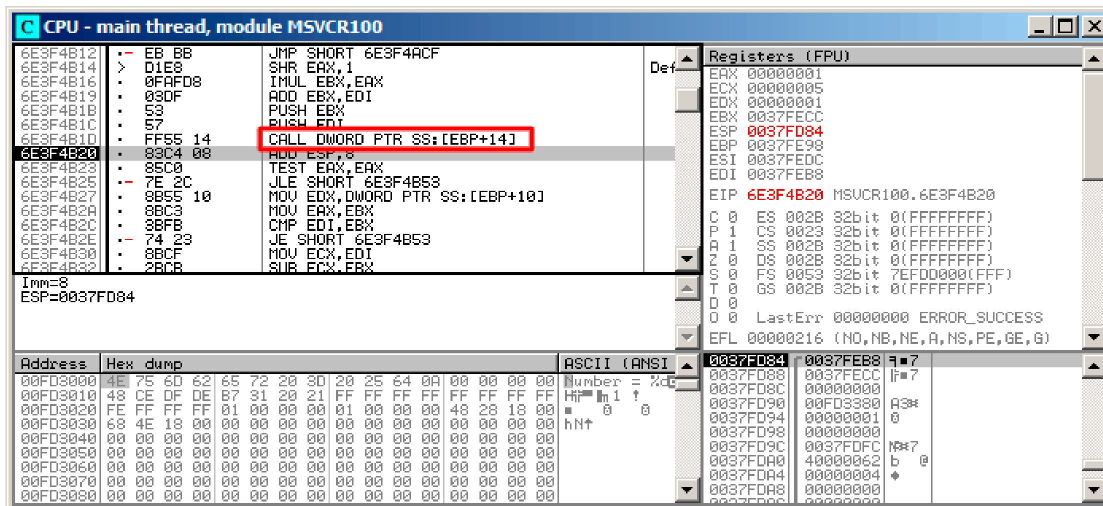


Abbildung 1.110: OllyDbg: der Code in `qsort()` direkt nach dem Aufruf von `comp()`

Das war beispielhaft ein Aufruf der Vergleichsfunktion.

Hier ist noch ein Screenshot von dem Moment des zweiten Aufrufs von comp()—nun müssen die verglichenen Werte andere sein:

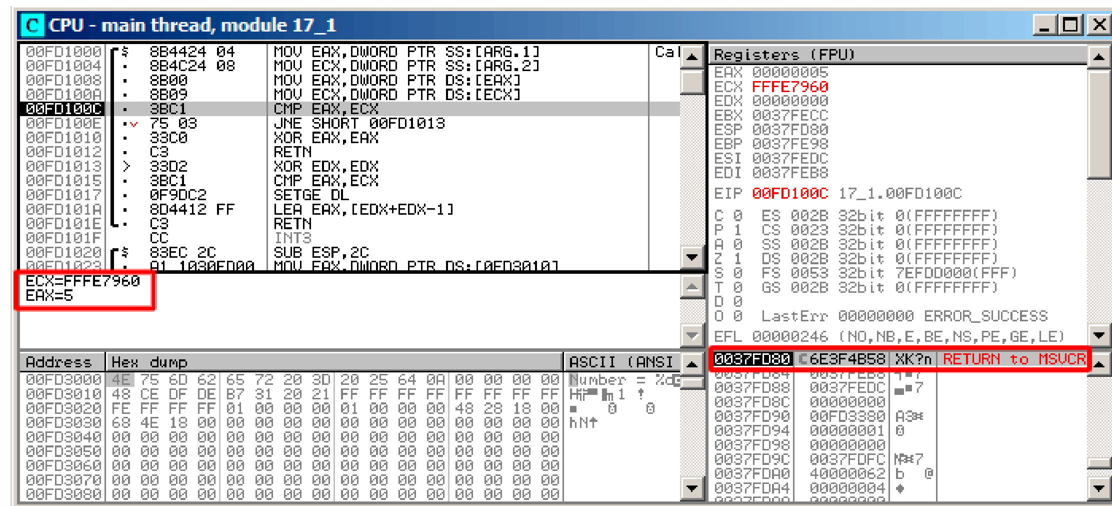


Abbildung 1.111: OllyDbg: zweiter Aufruf von comp()

MSVC + Tracer

Schauen wir uns auch an, welche Paare verglichen werden. Diese 10 Zahlen werden sortiert: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

Wir erhalten die Adressen des ersten CMP Befehls in comp(), d.i. 0x0040100C und wir haben hier einen Breakpoint gesetzt:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Jetzt erhalten wir Informationen über die Register am Breakpoint:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Filtern wir EAX und ECX heraus, so erhalten wir:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

Das entspricht 34 Paaren. Der Quicksort-Algorithmus benötigt hier also 34 Vergleichsoperationen um die 10 Zahlen zu sortieren.

MSVC + Tracer (Codebetrachtung)

Wir können auch alle möglichen Registerwerte über die Ablaufverfolgung sammeln und in IDA anzeigen.

Verfolgen wir also alle Befehle in comp():

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

Wir erhalten ein .idc-Skript und laden dieses in IDA:

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008                 mov     eax, [eax]      ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100A                 mov     ecx, [ecx]      ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100C                 cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                             ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx        ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017                 setnl   dl              ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                    ; EAX=1, 0xffffffff
.text:0040101E PtFuncCompare   endp
.text:0040101E
```

Abbildung 1.112: Tracer und IDA. Werte teilweise rechts abgeschnitten

IDA hat der Funktion einen Namen gegeben (PtFuncCompare)—denn IDA erkennt, dass der Pointer auf diese Funktion an qsort() übergeben wird.

Wir sehen, dass die Pointer a und b auf diverse Stellen im Array zeigen, aber die Schrittweite ist stets 4, da im Array 32-Bit-Werte gespeichert sind.

Wir sehen auch, dass die Befehle an den Stellen 0x401010 und 0x401012 nie ausgeführt wurden (deshalb wurden sie weiß gelassen). Da liegt daran, dass comp() nie 0 zurückgegeben hat, da sich im Array keine zwei gleichen Elemente befinden.

1.25.2 GCC

Kein großer Unterschied:

Listing 1.341: GCC

```
                lea     eax, [esp+40h+var_28]
                mov     [esp+40h+var_40], eax
                mov     [esp+40h+var_28], 764h
                mov     [esp+40h+var_24], 2Dh
```

```
                mov     [esp+40h+var_20], 0C8h
                mov     [esp+40h+var_1C], 0FFFFFF9Eh
                mov     [esp+40h+var_18], 0FF7h
                mov     [esp+40h+var_14], 5
                mov     [esp+40h+var_10], 0FFFFCFC7h
                mov     [esp+40h+var_C], 43Fh
                mov     [esp+40h+var_8], 58h
                mov     [esp+40h+var_4], 0FFFE7960h
                mov     [esp+40h+var_34], offset comp
                mov     [esp+40h+var_38], 4
                mov     [esp+40h+var_3C], 0Ah
                call    _qsort
```

Die Funktion comp():

```
                public comp
comp            proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                mov     edx, [eax]
                xor     eax, eax
                cmp     [ecx], edx
                jnz     short loc_8048458
                pop     ebp
                retn
loc_8048458:
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

Die Implementierung von qsort() befindet sich in libc.so.6 und ist eigentlich nur ein Wrapper für qsort_r().

Sie ruft quicksort() auf, wo unsere selbst definierte Funktion über einen übergebenen Pointer aufgerufen wird:

Listing 1.342: (file libc.so.6, glibc version—2.10.1)

```
...
.text:0002DDF6                 mov     edx, [ebp+arg_10]
.text:0002DDF9                 mov     [esp+4], esi
.text:0002DDFD                 mov     [esp], edi
.text:0002DE00                 mov     [esp+8], edx
.text:0002DE04                 call    [ebp+arg_C]
...
```

GCC + GDB (mit Quellcode)

Natürlich kennen wir den C-Quellcode unseres Beispiels (1.25 on page 455), sodass wir einen Breakpoint (b) auf die Zeile (11–die Zeile, in der der erste Vergleich auftaucht), setzen können. Wir müssen das Beispiel mit hinzugefügten Debugging-Informationen kompilieren (-g), sodass die Tabelle mit den Adressen und zugehörigen Zeilennummern verfügbar ist.

Wir können mit den Variablennamen (p) die Werte auch ausgeben: Die Debugging-Informationen verraten uns auch, welche Register und/oder Elemente auf dem lokalen Stack welche Variablen enthalten.

Wir sehen auch den Stack (bt) und können folgern, dass es eine Zwischenfunktion msort_with_tmp() in Glibc geben muss.

Listing 1.343: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11        if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11        if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0⤦
    ↪ xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↪ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry⤦
    ↪ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort⤦
    ↪ .c:45
```

```
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↙
    ↳ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ↙
    ↳ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=↙
    ↳ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp↙
    ↳ >) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

GCC + GDB (ohne Quellcode)

Oft gibt es aber keinen Quellcode, sodass wir die `comp()` Funktion (`disas`) disassemblieren und den ersten `CMP` Befehl finden müssen und dann an dieser Adresse einen Breakpoint (b) setzen.

An jedem Breakpoint werden wir alle Registerinhalte (`info registers`) in den Dump schreiben. Die Informationen zum Stack (`bt`) sind ebenfalls verfügbar, aber nur teilweise vollständig: es gibt keine Informationen zu den Zeilennummern in `comp()`.

Listing 1.344: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols ↙
    ↳ found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
   0x08048453 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:     mov    DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:    mov    edx,DWORD PTR [eax]
   0x08048464 <+23>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:    mov    eax,DWORD PTR [eax]
   0x08048469 <+28>:    cmp    edx,eax
   0x0804846b <+30>:    jne    0x8048474 <comp+39>
   0x0804846d <+32>:    mov    eax,0x0
   0x08048472 <+37>:    jmp    0x804848e <comp+65>
   0x08048474 <+39>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:    mov    edx,DWORD PTR [eax]
   0x08048479 <+44>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47>:    mov    eax,DWORD PTR [eax]
```

```
   0x0804847e <+49>:    cmp    edx,eax
   0x08048480 <+51>:    jge    0x8048489 <comp+60>
   0x08048482 <+53>:    mov    eax,0xffffffff
   0x08048487 <+58>:    jmp    0x804848e <comp+65>
   0x08048489 <+60>:    mov    eax,0x1
   0x0804848e <+65>:    leave
   0x0804848f <+66>:    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8       -1073745672
edx            0x764    1892
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff0fc       -1073745668
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104       -1073745660
edx            0xffffff9e       -98
ebx            0xb7fc0000       -1208221696
esp            0xbfffee58       0xbfffee58
ebp            0xbfffee68       0xbfffee68
esi            0xbffff108       -1073745656
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
```

```
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e       -98
ecx            0xbffff100       -1073745664
edx            0xc8     200
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff104       -1073745660
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0↙
    ↳ xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort↙
    ↳ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↙
    ↳ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort↙
    ↳ .c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↙
    ↳ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ↙
    ↳ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=↙
    ↳ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp↙
    ↳ >) at msort.c:307
#9  0x0804850d in main ()
```

1.25.3 Gefahr von Pointern auf Funktionen

Wie wir sehen können, erwartet die Funktion `qsort()` einen Pointer auf eine Funktion, die zwei *void** Argumente nimmt und einen Integer zurückgibt. Wenn man viele Vergleichsfunktionen im Code finden (eine vergleicht Strings, ein andere Integer, etc.) kommt man hier leicht durcheinander. Man könnte versuchen, ein Array aus Strings mit der Vergleichsfunktion für Integer zu sortieren und der Compiler würde bei diesem Bug nicht einmal eine Warnung ausgeben.

1.26 64-Bit-Werte in 32-Bit-Umgebungen

In einer 32-Bit-Umgebung sind GPR 32 Bit groß. Also werden 64-Bit-Werte in 32-Bit-Wertepaaren gespeichert und übergeben[152].

1.26.1 Rückgabe von 64-Bit-Werten

```
#include <stdint.h>

uint64_t f ()
{
        return 0x1234567890ABCDEF;
};
```

x86

In einer 32-Bit-Umgebung werden 64-Bit-Werte von Funktionen über das Registerpaar EDX:EAX zurückgegeben:

Listing 1.345: Optimierender MSVC 2010

```
_f      PROC
        mov     eax, -1867788817 ; 90abcdefH
        mov     edx, 305419896   ; 12345678H
        ret     0
_f      ENDP
```

ARM

Ein 64-Bit-Wert wird über das R0-R1 Registerpaar zurückgegeben (R1 enthält dabei den höheren und R0 den niederen Teil):

Listing 1.346: Optimierender Keil 6/2013 (ARM Modus)

```
||f|| PROC
        LDR      r0,|L0.12|
        LDR      r1,|L0.16|
        BX       lr
        ENDP

|L0.12|
        DCD      0x90abcdef
|L0.16|
        DCD      0x12345678
```

[152]Übrigens, 32-Bit-Werte werden als Paare in 16–Bit-Umgebungen auf der gleiche Art übergeben: **??** on page ??

MIPS

Ein 64-Bit-Wert wird über das V0-V1 ($2-$3) Registerpaar zurückgegeben (V0 ($2) enthält dabei den höheren und V1 ($3) den niederen Teil):

Listing 1.347: Optimierender GCC 4.4.5 (assembly listing)

```
        li      $3,-1867841536   # 0xffffffff90ab0000
        li      $2,305397760     # 0x12340000
        ori     $3,$3,0xcdef
        j       $31
        ori     $2,$2,0x5678
```

Listing 1.348: Optimierender GCC 4.4.5 (IDA)

```
       lui     $v1, 0x90AB
       lui     $v0, 0x1234
       li      $v1, 0x90ABCDEF
       jr      $ra
       li      $v0, 0x12345678
```

1.26.2 Übergabe von Argumenten bei Addition und Subtraktion

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
        return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
        return a-b;
};
```

x86

Listing 1.349: Optimierender MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f_add  PROC
```

```
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f_add  ENDP

_f_add_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f_add
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f_add_test ENDP

_f_sub  PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f_sub  ENDP
```

Wir sehen, dass in der Funktion `f_add_test()` jeder 64-Bit-Wert über zwei 32-Bit-Werten übergeben wird: zuerst der höhere Teil, dann der niedere Teil.

Addition und Subtraktion werden auch mit Paaren ausgeführt.

Bei einer Addition werden die niederen 32-Bit-Teile zuerst addiert. Tritt hierbei ein Übertrag auf, wird das CF Flag gesetzt.

Der folgende ADC Befehl addiert die höheren Teile der Operanden und addiert 1, falls $CF = 1$.

Subtraktion wird auch mit den Wertepaaren durchgeführt. Das erste SUB setzt ggf. das CF Flag, das von dem folgenden SBB Befehl geprüft wird: wenn das Carryflag gesetzt ist, wird am Ende 1 vom Ergebnis abgezogen.

Man erkennt im Code leicht, wie das Ergebnis der Funktion `f_add()` an `printf()` übergeben wird.

Listing 1.350: GCC 4.8.1 -O1 -fno-inline

```
_f_add:
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret
```

```
_f_add_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f_add
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\n"
        call    _printf
        add     esp, 28
        ret

_f_sub:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

Der Code von GCC ist identisch.

ARM

Listing 1.351: Optimierender Keil 6/2013 (ARM Modus)

```
f_add PROC
        ADDS     r0,r0,r2
        ADC      r1,r1,r3
        BX       lr
        ENDP

f_sub PROC
        SUBS     r0,r0,r2
        SBC      r1,r1,r3
        BX       lr
        ENDP

f_add_test PROC
        PUSH     {r4,lr}
        LDR      r2,|L0.68| ; 0x75939f79
        LDR      r3,|L0.72| ; 0x00001555
        LDR      r0,|L0.76| ; 0x73ce2ff2
        LDR      r1,|L0.80| ; 0x00000b3a
        BL       f_add
        POP      {r4,lr}
        MOV      r2,r0
        MOV      r3,r1
        ADR      r0,|L0.84| ; "%I64d\n"
        B        __2printf
        ENDP
```

```
|L0.68|
        DCD      0x75939f79
|L0.72|
        DCD      0x00001555
|L0.76|
        DCD      0x73ce2ff2
|L0.80|
        DCD      0x00000b3a
|L0.84|
        DCB      "%I64d\n",0
```

Der erste 64-Bit-Wert wird über das Registerpaar R0 und R1 übergeben, der zweite über R2 und R3. ARM verfügt ebenfalls über die Befehle ADC und SBC (die das Carryflag beachten). Man beachte: wenn die niederen Teile addiert bzw. subtrahiert werden, werden ADDS und SUBS Befehle mit dem Suffy -S verwendet. Dieser Suffix steht für „set flags“ (dt. setze Flags) und wird von den folgenden ADC/SBC Befehlen unbedingt benötigt. Würden die Flags nicht weiter beachtet werden, könnten hier ADD und SUB verwendet werden.

MIPS

Listing 1.352: Optimierender GCC 4.4.5 (IDA)

```
f_add:
; $a0 - höherer Teil von a
; $a1 - niederer Teil von a
; $a2 - höherer Teil von b
; $a3 - niederer Teil von b
                addu    $v1, $a3, $a1 ; addiere niedere Teile
                addu    $a0, $a2, $a0 ; addiere höhere Teile
; wird Übertrag durch Addition der niederen Teile erzeugt?
; falls ja, setze $v0 auf 1
                sltu    $v0, $v1, $a3
                jr      $ra
; add 1 to high part of result if carry should be generated:
                addu    $v0, $a0 ; branch delay slot
; $v0 - höherer Teil des Ergebnisses
; $v1 - niederer Teil des Ergebnisses

f_sub:
; $a0 - höherer Teil von a
; $a1 - niederer Teil von a
; $a2 - höherer Teil von b
; $a3 - niederer Teil von b
                subu    $v1, $a1, $a3 ; subtrahiere niedere Teile
                subu    $v0, $a0, $a2 ; subtrahiere höhere Teile
; wird Übertrag durch Subtrahieren der niederen Teile erzeugt?
; falls ja, setze $a0 auf 1
                sltu    $a1, $v1
                jr      $ra
; subtrahiere 1 vom höheren Teil des Ergebnisses, falls Übertrag vorliegt:
```

```
                subu    $v0, $a1 ; branch delay slot
; $v0 - höherer Teil des Ergebnisses
; $v1 - niederer Teil des Ergebnisses

f_add_test:

var_10          = -0x10
var_4           = -4

                lui     $gp, (__gnu_local_gp >> 16)
                addiu   $sp, -0x20
                la      $gp, (__gnu_local_gp & 0xFFFF)
                sw      $ra, 0x20+var_4($sp)
                sw      $gp, 0x20+var_10($sp)
                lui     $a1, 0x73CE
                lui     $a3, 0x7593
                li      $a0, 0xB3A
                li      $a3, 0x75939F79
                li      $a2, 0x1555
                jal     f_add
                li      $a1, 0x73CE2FF2
                lw      $gp, 0x20+var_10($sp)
                lui     $a0, ($LC0 >> 16)  # "%lld\n"
                lw      $t9, (printf & 0xFFFF)($gp)
                lw      $ra, 0x20+var_4($sp)
                la      $a0, ($LC0 & 0xFFFF)  # "%lld\n"
                move    $a3, $v1
                move    $a2, $v0
                jr      $t9
                addiu   $sp, 0x20

$LC0:           .ascii "%lld\n"<0>
```

MIPS besitzt kein Register für die Flags, sodass keine derartige Information nach der Ausführung von arithmetischen Operationen verfügbar ist. Es gibt also keine Befehle wie ADC oder SBB in x86. Um zu prüfen, ob das Carryflag gesetzt werden muss, wird ein SLTU Befehl verwendet, der das Zielregister auf 1 oder 0 setzt. Diese 1 oder 0 wird dann zum Ergebnis addiert bzw. davon subtrahiert.

1.26.3 Multiplikation und Division

```
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
        return a/b;
};
```

```
uint64_t f_rem (uint64_t a, uint64_t b)
{
        return a % b;
};
```

x86

Listing 1.353: Optimierender MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __allmul ; long long Multiplikation
        pop     ebp
        ret     0
_f_mul  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aulldiv ; long long Division ohne Vorzeichen
        pop     ebp
        ret     0
_f_div  ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _b$[ebp+4]
        push    eax
```

```
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp+4]
        push    edx
        mov     eax, DWORD PTR _a$[ebp]
        push    eax
        call    __aullrem ; long long Rest ohne Vorzeichen
        pop     ebp
        ret     0
_f_rem  ENDP
```

Da Multiplikation und Division komplexere Operationen sind, verwendet der Compiler für deren Ausführung in der Regel Bibliotheksfunktionen.

Diese Funktionen werden hier beschrieben:.3 on page 685.

Listing 1.354: Optimierender GCC 4.8.1 -fno-inline

```
_f_mul:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f_div:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___udivdi3 ; Division ohne Vorzeichen
        add     esp, 28
        ret

_f_rem:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
```

```
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___umoddi3 ; Modulo ohne Vorzeichen
        add     esp, 28
        ret
```

GCC verhält sich wie erwartet, aber der Code für die Multiplikation wird direkt in der Funktion platziert und ist so effizienter. In GCC haben die Bibliotheksfunktionen andere Namen: .2 on page 685.

ARM

Keil für Thumb mode fügt Aufrufe von Routinen aus Bibliotheken ein:

Listing 1.355: Optimierender Keil 6/2013 (Thumb Modus)

```
||f_mul|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_lmul
        POP      {r4,pc}
        ENDP

||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOVS     r0,r2
        MOVS     r1,r3
        POP      {r4,pc}
        ENDP
```

Keil für ARM mode ist wiederum in der Lage Code für 64-Bit-Multiplikation zu erzeugen:

Listing 1.356: Optimierender Keil 6/2013 (ARM Modus)

```
||f_mul|| PROC
        PUSH     {r4,lr}
        UMULL    r12,r4,r0,r2
        MLA      r1,r2,r1,r4
        MLA      r1,r0,r3,r1
        MOV      r0,r12
        POP      {r4,pc}
        ENDP

||f_div|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
```

```
        POP      {r4,pc}
        ENDP

||f_rem|| PROC
        PUSH     {r4,lr}
        BL       __aeabi_uldivmod
        MOV      r0,r2
        MOV      r1,r3
        POP      {r4,pc}
        ENDP
```

MIPS

Der optimierende GCC für MIPS kann Code für 64-Bit-Multiplikation erzeugen, muss aber für die Division auf eine Programmbibliothek zurückgreifen:

Listing 1.357: Optimierender GCC 4.4.5 (IDA)

```
f_mul:
       mult    $a2, $a1
       mflo    $v0
       or      $at, $zero  ; NOP
       or      $at, $zero  ; NOP
       mult    $a0, $a3
       mflo    $a0
       addu    $v0, $a0
       or      $at, $zero  ; NOP
       multu   $a3, $a1
       mfhi    $a2
       mflo    $v1
       jr      $ra
       addu    $v0, $a2

f_div:

var_10 = -0x10
var_4  = -4

       lui     $gp, (__gnu_local_gp >> 16)
       addiu   $sp, -0x20
       la      $gp, (__gnu_local_gp & 0xFFFF)
       sw      $ra, 0x20+var_4($sp)
       sw      $gp, 0x20+var_10($sp)
       lw      $t9, (__udivdi3 & 0xFFFF)($gp)
       or      $at, $zero
       jalr    $t9
       or      $at, $zero
       lw      $ra, 0x20+var_4($sp)
       or      $at, $zero
       jr      $ra
       addiu   $sp, 0x20

f_rem:
```

```
var_10 = -0x10
var_4  = -4

       lui     $gp, (__gnu_local_gp >> 16)
       addiu   $sp, -0x20
       la      $gp, (__gnu_local_gp & 0xFFFF)
       sw      $ra, 0x20+var_4($sp)
       sw      $gp, 0x20+var_10($sp)
       lw      $t9, (__umoddi3 & 0xFFFF)($gp)
       or      $at, $zero
       jalr    $t9
       or      $at, $zero
       lw      $ra, 0x20+var_4($sp)
       or      $at, $zero
       jr      $ra
       addiu   $sp, 0x20
```

Hier gibt es eine Menge NOPs; möglicherweise sind dies delay slots, die nach dem Multiplikationsbefehl eingefügt wurden (diese Vorgehensweise ist jedoch langsamer als andere Befehle).

1.26.4 Verschiebung nach rechts

```
#include <stdint.h>

uint64_t f (uint64_t a)
{
        return a>>7;
};
```

x86

Listing 1.358: Optimierender MSVC 2012 /Ob1

```
_a$ = 8         ; size = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f      ENDP
```

Listing 1.359: Optimierender GCC 4.8.1 -fno-inline

```
_f:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
        ret
```

Das Verschieben geschieht ebenfalls zweigeteilt: zunächst wird der niedere Teil verschoben, danach der höhere. Der niedere Teil wird mithilfe des Befehls SHRD verschoben; er verschiebt den Wert in EAX um 7 Bits, holt aber die nachrutschenden Bits aus EDX, d.h. aus dem höheren Teil. Mit anderen Worten: der 64-Bit-Wert aus EDX:EAX wird als ganzes um 7 Bits verschoben und die niederen 32 Bits des Ergebnisses werden in EAX abgelegt. Der höhere Teil wird mit dem häufig verwendeten SHR Befehl verschoben, da die frei werdenden Bits im höheren Teil mit Nullen aufgefüllt werden müssen.

ARM

ARM verfügt im Gegensatz zu x86 nicht über einen SHRD Befehl, sodass der Keil Compiler die Aufgabe mit einer Kombination aus einfachen Schiebebefehlen und OR-Operationen durchführen muss:

Listing 1.360: Optimierender Keil 6/2013 (ARM Modus)

```
||f|| PROC
        LSR      r0,r0,#7
        ORR      r0,r0,r1,LSL #25
        LSR      r1,r1,#7
        BX       lr
        ENDP
```

Listing 1.361: Optimierender Keil 6/2013 (Thumb Modus)

```
||f|| PROC
        LSLS     r2,r1,#25
        LSRS     r0,r0,#7
        ORRS     r0,r0,r2
        LSRS     r1,r1,#7
        BX       lr
        ENDP
```

MIPS

GCC für MIPS folgt dem gleichen Algorithmus wie Keil für Thumb mode:

Listing 1.362: Optimierender GCC 4.4.5 (IDA)

```
f:
       sll     $v0, $a0, 25
       srl     $v1, $a1, 7
       or      $v1, $v0, $v1
       jr      $ra
       srl     $v0, $a0, 7
```

1.26.5 32-Bit-Werte in 64-Bit-Werte umwandeln

```
#include <stdint.h>

int64_t f (int32_t a)
{
        return a;
};
```

x86

Listing 1.363: Optimierender MSVC 2012

```
_a$ = 8
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP
```

Hier müssen wir einen vorzeichenbehafteten 32-Bit-Wert in einen 64-Bit-Wert mit Vorzeichen umwandeln. Vorzeichenlose Werte werden ganz einfach umgewandelt: alle Bits im höheren Teil werden auf 0 gesetzt. Bei Werten mit Vorzeichen funktioniert diese Methode nicht: das Vorzeichen muss in den höheren Teil des 32-Bit-Ergebnisses kopiert werden. Der Befehl CDQ übernimmt hier diese Aufgabe. Er nimmt seinen Eingabewert aus EAX, erweitert ihn auf 64 Bit und speichert das Ergebnis im Registerpaar EDX/EAX. Mit anderen Worten: CDQ extrahiert das Vorzeichen aus EAX (dies ist das MSB in EAX) und setzt je nach Wert des Vorzeichens alle 32 Bits in EDX auf 0 oder 1. Dieser Befehl ähnelt dem Befehl MOVSX.

ARM

Listing 1.364: Optimierender Keil 6/2013 (ARM Modus)

```
||f|| PROC
        ASR      r1,r0,#31
        BX       lr
        ENDP
```

Keil für ARM geht anders vor: er verschiebt den Eingabewert um 31 Bits nach rechts. Wie wir wissen ist das Vorzeichenbit das **MSB!** und die arithmetische Verschiebung kopiert das Vorzeichenbit in die ausgeschobenen Bits. Nach Ausführung von „ASR r1,r0,#31“ enthält R1 also 0xFFFFFFFF, falls der Eingabewert negativ war und ansonsten 0. R1 enthält den höheren Teil des resultierenden 64-Bit-Wertes. Mit anderen Worten: dieser Code kopiert das **MSB!** (Vorzeichenbit) vom Eingabewert in R0 in alle Bits der höheren 32 Bits des Ergebnisses.

MIPS

GCC für MIPS erzeugt das gleich wie Keil für ARM mode:

Listing 1.365: Optimierender GCC 4.4.5 (IDA)

```
f:
    sra     $v0, $a0, 31
    jr      $ra
    move    $v1, $a0
```

1.27 SIMD

SIMD ist ein Akronym: *Single Instruction, Multiple Data*. Wie der Name schon sagt, verarbeitet SIMD mehrere Daten in nur einem Befehl.

Wie die FPU sieht das CPU-Subsystem wie ein separater Prozessor innerhalb von x86 aus.

SIMD begann als MMX in x86. 8 neue 64-Bit-Register erschienen: MM0-MM7.

Jedes MMX Register kann 2 32-Bit-Werte, 4 16-Bit-Werte oder 8 Byte aufnehmen. Es ist zum Beispiel möglich, 8 8-Bit-Werte gleichzeitig zu addieren, indem zwei Werte in MMX Registern addiert werden.

Ein einfaches Beispiel ist ein Graphikeditor, der ein Bild als ein zweidimensionales Array darstellt. Wenn der User die Helligkeit des Bildes verändert, muss der Editor einen bestimmten Koeffizienten zu jedem Pixelwert addieren bzw. von ihm subtrahieren. Wenn wir um es einfach zu halten annehmen, dass das Bild in schwarzweiß ist und jeder Pixel durch ein Byte definiert ist, dann ist es möglich die Helligkeit von 8 Pixeln gleichzeitig zu verändern.

Das ist übrigens der Grund dafür, dass es in SIMB die Sättigungsbefehle gibt.

Wenn der User die Helligkeit im Graphikeditor verändert, sind Überlauf und Unterlauf nicht erwünscht, weshalb es in SIMD Additionsbefehle gibt, die nicht weiter addieren, wenn der Maximalwert bereits erreicht its, etc.

Als MMX erschien befanden sich diese Register räumlich innerhalb der FPU Register. Es war möglich entweder die FPU oder MMX zur gleichen Zeit zu benutzen. Man könnte meinen, dass Intel dieses Layout gewählt hat um Transistoren zu sparen, aber der wirkliche Grund für diese Symbiose ist trivialer —ältere Betriebssysteme, die die zusätzlichen CPU Register nicht erwarteten, hätten deren Inhalte nicht gesichert, sicherten aber sehr wohl den Inhalt der FPU Register. Dadurch funktioniert die Kombination aus MMX CPU und altem Betriebssystem, wenn MMX Features verwendet werden.

SSE ist eine Erweiterung der SIMB Register auf 128 Bit. Die Register sind hier von der FPU getrennt.

AVX ist eine andere Erweiterung–auf 256 Bits. Kommen wir nun zur Verwendung in der Praxis. Natürlich gibt es Funktionen zum Kopieren (`memcpy`) oder Vergleichen (`memcmp`) von Speicherblöcken etc.

Ein weiteres Beispiel: der DES-Verschlüsselungsalgorithmus nimmt einen 64-Bit-Block und einen 56-Bit-Key, verschlüsselt den Block und erzeugt ein 64-Bit-Ergebnis. Der

DES-Algorithmus kann als sehr großer Schaltkreis aus Drähten und AND/OR/NOT-Gattern aufgefasst werden.

Hinter Bitslice DES[153] —steckt die Idee, mehrere Gruppen von Blöcken und Schlüsseln simultan zu verarbeiten. Eine Variable vom Typ *unsigned int* umfasst in x86 beispielsweise 32 Bit, sodass es möglich ist, in ihr die Zwischenergebnisse für 32 Block-Schlüssel-Paare gleichzeitig zu speichern, indem $64+56$ Variablen vom Typ *unsigned int* verwendet werden.

Es gibt es Tool um Oracle RDBMS Passwörter und Hashes (die auf DES basieren) per Brute-Force zu knacken. Hierbei wird ein nur wenig veränderter Bitslice-DES-Algorithmus für SSE2 und AVX verwendet—nun ist es möglich, 128 oder 256 Block-Schlüssel-Paare simultan zu verschlüsseln.

http://conus.info/utils/ops_SIMD/

1.27.1 Vektorisierung

Vektorisierung[154] tritt auf, wenn man beispielsweise eine Schleife hat, die mehrere Arrays als Input und ein Array als Output hat. Der Rumpf der Schleife nimmt Werte aus den Eingabearrays, vearbeitet sie und legt das Ergebnis im Outputarray ab. Vektorisierung wird hier verwendet um mehrere Elemente gleichzeitig zu verarbeiten.

Vektorisierung ist keine besonders neue Idee: der Autor hat sie bereits auf dem Cray Y-MP Supercomputer aus dem Jahre 1988 ausgemacht, als er mit dessen kleinerem Brude Cray Y-MP EL spielte.

Vectorization is not very fresh technology: the author of this textbook saw it at least on the Cray Y-MP supercomputer line from 1988 when he played with its „lite" version Cray Y-MP EL [155].

Ein Beispiel:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Dieses Codefragment nimmt Elemente aus A und B, multipliziert sie und speichert das Ergebnis in C.

Wenn jedes Arrayelement ein 32-Bit *int* ist, ist es möglich 4 Elemente aus A in ein 128-Bit-XMM-Register zu laden, 4 weitere Elemente aus B in ein anderes, und dann durch die Befehle *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) und PMULHW (*Multiply Packed Signed Integers and Store High Result*) 4 64-Bit-Produkte auf einmal zu erzeugen.

Dadurch wir der Rumpf der Schleife nur $\frac{1024}{4} = 256$ Mal ausgeführt. Das ist nur ein Viertel der normalen Anzahl an Durchläufen und geht natürlich schneller.

[153] http://www.darkside.com.au/bitslice/
[154] Wikipedia: vectorization
[155] Er befindet sich im Museum für Supercomputer: http://www.cray-cyber.org

Beispiel zur Addition

Manche Compiler können Vektorsierung in einfachen Fällen automatisch anwenden, zum Beispiel Intel C++[156].

Hier ist eine kleine Funktion:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];

        return 0;
};
```

Intel C++

Kompilieren wir das Programm mit Intel C++ 11.1.051 win32:

`icl intel.cpp /QaxSSE2 /Faintel.asm /Ox`

Wir erhalten (in IDA) folgenden Code:

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr  0Ch
ar3    = dword ptr  10h

       push    edi
       push    esi
       push    ebx
       push    esi
       mov     edx, [esp+10h+sz]
       test    edx, edx
       jle     loc_15B
       mov     eax, [esp+10h+ar3]
       cmp     edx, 6
       jle     loc_143
       cmp     eax, [esp+10h+ar2]
       jbe     short loc_36
       mov     esi, [esp+10h+ar2]
       sub     esi, eax
       lea     ecx, ds:0[edx*4]
       neg     esi
       cmp     ecx, esi
       jbe     short loc_55
```

[156] Mehr zur automatischen Vektorsierung in Intel C++ unter: Auszug: Effektive automatische Vektorisierung

```
loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax        ; edi = ar3
        and     edi, 0Fh        ; liegt ar3 auf 16-Byte-Grenze?
        jz      short loc_9A    ; ja
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi
```

```
loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
       mov     edx, [ecx+esi*4]
       add     edx, [ebx+esi*4]
       mov     [eax+esi*4], edx
       inc     esi
       cmp     esi, edi
       jb      short loc_C1
       mov     ecx, [esp+10h+var_10]
       mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
       mov     esi, [esp+10h+ar2]
       lea     esi, [esi+edi*4] ; liegt ar2+i*4 auf 16-Byte-Grenze?
       test    esi, 0Fh
       jz      short loc_109   ; yes!
       mov     ebx, [esp+10h+ar1]
       mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
       movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
       movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 nicht liegt auf
    16-Byte-Grenze: lade es nach XMM0
       paddd   xmm1, xmm0
       movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
       add     edi, 4
       cmp     edi, ecx
       jb      short loc_ED
       jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
       mov     ebx, [esp+10h+ar1]
       mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
       movdqu  xmm0, xmmword ptr [ebx+edi*4]
       paddd   xmm0, xmmword ptr [esi+edi*4]
       movdqa  xmmword ptr [eax+edi*4], xmm0
       add     edi, 4
       cmp     edi, ecx
       jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
         ; f(int,int *,int *,int *)+164
       cmp     ecx, edx
       jnb     short loc_15B
       mov     esi, [esp+10h+ar1]
       mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
       mov     ebx, [esi+ecx*4]
       add     ebx, [edi+ecx*4]
       mov     [eax+ecx*4], ebx
       inc     ecx
       cmp     ecx, edx
```

```
       jb      short loc_133
       jmp     short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
         ; f(int,int *,int *,int *)+3A ...
       mov     esi, [esp+10h+ar1]
       mov     edi, [esp+10h+ar2]
       xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
       mov     ebx, [esi+ecx*4]
       add     ebx, [edi+ecx*4]
       mov     [eax+ecx*4], ebx
       inc     ecx
       cmp     ecx, edx
       jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
         ; f(int,int *,int *,int *)+129 ...
       xor     eax, eax
       pop     ecx
       pop     ebx
       pop     esi
       pop     edi
       retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
         ; f(int,int *,int *,int *)+9F
       xor     ecx, ecx
       jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

Die SSE2-Befehle sind:

- MOVDQU (*Move Unaligned Double Quadword*)—lädt 16 Byte aus dem Speicher in ein XMM-Register.
- PADDD (*Add Packed Integers*) addiert 4 Paare von 32-Bit-Zahlen und speichert das Ergebnis im ersten Operanden. Hier wird übrigens bei einem Überlauf keine Exception geworfen und auch keine Flags gesetzt, aber es werden dann nur die niederen 23 Bit des Ergebnisses gespeichert. Wenn einer der Operanden von PADDD die Speicheradresse eines Wertes ist, muss sich die Adresse auf einer 16-Byte-Grenze befinden. Ist dies nicht der Fall, wird eine Exception ausgelöst.
- MOVDQA (*Move Aligned Double Quadword*) entspricht MOVDQU, verlangt aber, dass die Adresse des Wertes im Speicher auf einer 16-Byte-Grenze liegt. Ist dies nicht der Fall, wird eine Exception ausgelöst. MOVDQA ist schneller als MOVDQU, hat aber die genannte zusätzliche Vorbedingung.

Diese SSE2-Befehle werden nur dann ausgeführt, wenn auf mehr als 4 Paaren gearbeitet werden muss und sich der Pointer ar3 auf einer 16-Byte-Grenze befindet.

Wenn sich ar2 ebenfalls auf einer 16-Byte-Grenze befindet, wird das folgende Codefragment ebenfalls ausgeführt:

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Ansonsten wird der Wert aus ar2 mit MOVDQU nach XMM0 geladen. MOVDQU benötigt keinen angeordneten Pointer, arbeitet aber möglicherweise langsamer:

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 liegt nicht auf einer
    16-Byte-Grenze: lade es nach XMM0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

In allen anderen Fällen wird nicht-SSE2-Code ausgeführt.

GCC

GCC kann in einfachen Fällen ebenfalls Vektorisierung durchführen[157], wenn die Option -O3 verwendet und SSE2 Unterstützung aktiviert ist: -msse2.

Wie erhalten den folgenden Code (GCC 4.4.1):

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8
```

[157] Mehr zur Unterstützung von Vektorisierung in GCC: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
```

```
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Fast der gleiche wie Intel C++, aber nicht ganz so kleinschrittig.

Beispiel zum Kopieren von Speicherblöcken

Betrachten wir erneut das Beispiel zum einfachen memcpy(1.16.2 on page 226):

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
        size_t i;
        for (i=0; i<cnt; i++)
                dst[i]=src[i];
};
```

Der optimierende GCC 4.9.1 erzeugt folgenden Code:

Listing 1.366: Optimierender GCC 4.9.1 x64

```
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
        test    rdx, rdx
        je      .L41
        lea     rax, [rdi+16]
        cmp     rsi, rax
        lea     rax, [rsi+16]
        setae   cl
        cmp     rdi, rax
        setae   al
        or      cl, al
        je      .L13
        cmp     rdx, 22
        jbe     .L13
        mov     rcx, rsi
        push    rbp
        push    rbx
        neg     rcx
        and     ecx, 15
        cmp     rcx, rdx
        cmova   rcx, rdx
        xor     eax, eax
        test    rcx, rcx
        je      .L4
        movzx   eax, BYTE PTR [rsi]
        cmp     rcx, 1
        mov     BYTE PTR [rdi], al
        je      .L15
        movzx   eax, BYTE PTR [rsi+1]
        cmp     rcx, 2
        mov     BYTE PTR [rdi+1], al
        je      .L16
        movzx   eax, BYTE PTR [rsi+2]
        cmp     rcx, 3
        mov     BYTE PTR [rdi+2], al
        je      .L17
        movzx   eax, BYTE PTR [rsi+3]
        cmp     rcx, 4
```

```
        mov     BYTE PTR [rdi+3], al
        je      .L18
        movzx   eax, BYTE PTR [rsi+4]
        cmp     rcx, 5
        mov     BYTE PTR [rdi+4], al
        je      .L19
        movzx   eax, BYTE PTR [rsi+5]
        cmp     rcx, 6
        mov     BYTE PTR [rdi+5], al
        je      .L20
        movzx   eax, BYTE PTR [rsi+6]
        cmp     rcx, 7
        mov     BYTE PTR [rdi+6], al
        je      .L21
        movzx   eax, BYTE PTR [rsi+7]
        cmp     rcx, 8
        mov     BYTE PTR [rdi+7], al
        je      .L22
        movzx   eax, BYTE PTR [rsi+8]
        cmp     rcx, 9
        mov     BYTE PTR [rdi+8], al
        je      .L23
        movzx   eax, BYTE PTR [rsi+9]
        cmp     rcx, 10
        mov     BYTE PTR [rdi+9], al
        je      .L24
        movzx   eax, BYTE PTR [rsi+10]
        cmp     rcx, 11
        mov     BYTE PTR [rdi+10], al
        je      .L25
        movzx   eax, BYTE PTR [rsi+11]
        cmp     rcx, 12
        mov     BYTE PTR [rdi+11], al
        je      .L26
        movzx   eax, BYTE PTR [rsi+12]
        cmp     rcx, 13
        mov     BYTE PTR [rdi+12], al
        je      .L27
        movzx   eax, BYTE PTR [rsi+13]
        cmp     rcx, 15
        mov     BYTE PTR [rdi+13], al
        jne     .L28
        movzx   eax, BYTE PTR [rsi+14]
        mov     BYTE PTR [rdi+14], al
        mov     eax, 15
.L4:
        mov     r10, rdx
        lea     r9, [rdx-1]
        sub     r10, rcx
        lea     r8, [r10-16]
        sub     r9, rcx
        shr     r8, 4
        add     r8, 1
```

```
        mov     r11, r8
        sal     r11, 4
        cmp     r9, 14
        jbe     .L6
        lea     rbp, [rsi+rcx]
        xor     r9d, r9d
        add     rcx, rdi
        xor     ebx, ebx
.L7:
        movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
        add     rbx, 1
        movups  XMMWORD PTR [rcx+r9], xmm0
        add     r9, 16
        cmp     rbx, r8
        jb      .L7
        add     rax, r11
        cmp     r10, r11
        je      .L1
.L6:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        lea     rcx, [rax+1]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+1+rax]
        mov     BYTE PTR [rdi+1+rax], cl
        lea     rcx, [rax+2]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+2+rax]
        mov     BYTE PTR [rdi+2+rax], cl
        lea     rcx, [rax+3]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+3+rax]
        mov     BYTE PTR [rdi+3+rax], cl
        lea     rcx, [rax+4]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+4+rax]
        mov     BYTE PTR [rdi+4+rax], cl
        lea     rcx, [rax+5]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+5+rax]
        mov     BYTE PTR [rdi+5+rax], cl
        lea     rcx, [rax+6]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+6+rax]
        mov     BYTE PTR [rdi+6+rax], cl
        lea     rcx, [rax+7]
        cmp     rdx, rcx
```

```
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+7+rax]
        mov     BYTE PTR [rdi+7+rax], cl
        lea     rcx, [rax+8]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+8+rax]
        mov     BYTE PTR [rdi+8+rax], cl
        lea     rcx, [rax+9]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+9+rax]
        mov     BYTE PTR [rdi+9+rax], cl
        lea     rcx, [rax+10]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+10+rax]
        mov     BYTE PTR [rdi+10+rax], cl
        lea     rcx, [rax+11]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+11+rax]
        mov     BYTE PTR [rdi+11+rax], cl
        lea     rcx, [rax+12]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+12+rax]
        mov     BYTE PTR [rdi+12+rax], cl
        lea     rcx, [rax+13]
        cmp     rdx, rcx
        jbe     .L1
        movzx   ecx, BYTE PTR [rsi+13+rax]
        mov     BYTE PTR [rdi+13+rax], cl
        lea     rcx, [rax+14]
        cmp     rdx, rcx
        jbe     .L1
        movzx   edx, BYTE PTR [rsi+14+rax]
        mov     BYTE PTR [rdi+14+rax], dl
.L1:
        pop     rbx
        pop     rbp
.L41:
        rep ret
.L13:
        xor     eax, eax
.L3:
        movzx   ecx, BYTE PTR [rsi+rax]
        mov     BYTE PTR [rdi+rax], cl
        add     rax, 1
        cmp     rax, rdx
        jne     .L3
        rep ret
.L28:
```

```
        mov     eax, 14
        jmp     .L4
.L15:
        mov     eax, 1
        jmp     .L4
.L16:
        mov     eax, 2
        jmp     .L4
.L17:
        mov     eax, 3
        jmp     .L4
.L18:
        mov     eax, 4
        jmp     .L4
.L19:
        mov     eax, 5
        jmp     .L4
.L20:
        mov     eax, 6
        jmp     .L4
.L21:
        mov     eax, 7
        jmp     .L4
.L22:
        mov     eax, 8
        jmp     .L4
.L23:
        mov     eax, 9
        jmp     .L4
.L24:
        mov     eax, 10
        jmp     .L4
.L25:
        mov     eax, 11
        jmp     .L4
.L26:
        mov     eax, 12
        jmp     .L4
.L27:
        mov     eax, 13
        jmp     .L4
```

1.27.2 SIMD `strlen()` Implementierung

Man beachte, dass die SIMD Befehle über spezielle Makros[158] in den C/C++ Code eingefügt werden können. Einige dieser Makros für MSVC befinden sich in der Datei `intrin.h`.

Es ist möglich die Funktion `strlen()`[159] mit SIMD Befehlen zu implementieren, so-

[158] MSDN: MMX, SSE, und SSE2 Intrinsics
[159] strlen() —Funktion der Standard-C-Bibliothek, mit der die Länge eines Strings berechnet wird

dass sie 2-2.5-mal schneller als die normale Implementierung arbeitet. Diese Funktion lädt 16 Zeichen in ein XMM-Register und prüft jedes auf Null.[160].

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (unsigned int)↙
    ↳ str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

Kompilieren wir das Beispiel in MSVC 2010 mit der Option /Ox:

Listing 1.367: Optimierender MSVC 2010

```
_pos$75552 = -4          ; size = 4
_str$ = 8                ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push     ebp
    mov      ebp, esp
    and      esp, -16            ; fffffff0H
    mov      eax, DWORD PTR _str$[ebp]
    sub      esp, 12             ; 0000000cH
    push     esi
    mov      esi, eax
    and      esi, -16            ; fffffff0H
    xor      edx, edx
```

[160]Dieses Beispiel basiert auf Quellcode von: http://www.strchr.com/sse2_optimised_strlen.

```
    mov      ecx, eax
    cmp      esi, eax
    je       SHORT $LN4@strlen_sse
    lea      edx, DWORD PTR [eax+1]
    npad     3 ; align next label
$LL11@strlen_sse:
    mov      cl, BYTE PTR [eax]
    inc      eax
    test     cl, cl
    jne      SHORT $LL11@strlen_sse
    sub      eax, edx
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
$LN4@strlen_sse:
    movdqa   xmm1, XMMWORD PTR [eax]
    pxor     xmm0, xmm0
    pcmpeqb  xmm1, xmm0
    pmovmskb eax, xmm1
    test     eax, eax
    jne      SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa   xmm1, XMMWORD PTR [ecx+16]
    add      ecx, 16                    ; 00000010H
    pcmpeqb  xmm1, xmm0
    add      edx, 16                    ; 00000010H
    pmovmskb eax, xmm1
    test     eax, eax
    je       SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf      eax, eax
    mov      ecx, eax
    mov      DWORD PTR _pos$75552[esp+16], eax
    lea      eax, DWORD PTR [ecx+edx]
    pop      esi
    mov      esp, ebp
    pop      ebp
    ret      0
?strlen_sse2@@YAIPBD@Z ENDP                ; strlen_sse2
```

Um die Funktionsweise zu verstehen müssen wir uns zunächst den Zweck der Funktion klarmachen. Sie berechnet die Länge eines C-Strings; anders ausgedrückt: sie sucht nach dem Nullbyte und berechnet dann dessen Position relativ zum Anfang des Strings.

Zunächst prüfen wir, ob der `str` Pointer auf einer 16-Byte-Grenze liegt. Wenn nicht, rufen wir die normale Implementierung von `strlen()` auf.

Danach laden wir die nächsten 16 Byte mit `MOVDQA` in das Register `XMM1`.

Der aufmerksame Leser fragt sich vielleicht, warum hier `MOVDQU` nicht verwendet wird, da dieser Befehl die Daten auch dann aus dem Speicher laden kann, wenn der Pointer nicht auf einer 16-Byte-Grenze liegt.

Man könnte es wie folgt lösen: wenn der Pointer entsprechend angeordnet ist, verwende MOVDQA zum Laden, ansonsten den langsameren Befehl MOVDQU.

Aber an dieser Stelle lauert ein weiterer Fallstrick:

In den Betriebssystemen der Reihe Windows NT (aber nicht nur dort) wird Speicher in Seiten von 4KiB (4096 Byte) reserviert. Jeder win32-Prozess hat 4GiB zur Verfügung, auch wenn tatsächlich nur einige Teile des Adressraumes wirklich mit dem physischen Speicher verbunden sind. Wenn der Prozess auf einen nicht vorhandenen Speicherblock zugreift, wird eine Exception ausgelöst. So arbeitet VM[161].

Eine Funktion, die 16 Byte auf einmal lädt, könnte also eine solche Grenze im Speicherblock übertreten. Nehmen wir an, das Betriebssystem hat 8192 (0x2000) Byte ab der Adresse 0x008c000 reserviert. Dieser Block umfasst also die Bytes von Adresse 0x08c0000 bis einschließlich 0x008c1fff.

Hinter diesen Block, d.h. ab Adresse 0x008c2000 befindet sich nichts, d.h. das Betriebssystem hat hier keinen weiteren Speicher reservietrt. Jeder Versuch auf diesen Speicherbereich zuzugreifen, wird eine Exception auslösen.

Betrachten wir weiter das Beispiel, in dem ein Programm einen String enthält, der fast am Ende des Blocks 5 Zeichen belegt. Das ist zunächst nichts Schlimmes.

0x008c1ff8	’h’
0x008c1ff9	’e’
0x008c1ffa	’l’
0x008c1ffb	’l’
0x008c1ffc	’o’
0x008c1ffd	’\x00’
0x008c1ffe	Zufallswert
0x008c1fff	Zufallswert

Unter normalen Umständen ruft das Programm `strlen()` auf und übergibt einen Pointer auf den String `'hello'`, der an der Speicheradresse 0x000c1ff0 liegt. `strlen()` liest ein Byte zur Zeit bis zur Adresse 0x000c1ffd, an der sich ein Nullbyte befindet, und hält dann an.

Wenn unser `strlen()` 16 Byte auf einmal von irgendeiner Startadresse aus liest, richtig angeordnet oder nicht, könnte MOVDQU versuchen 16 Byte auf einmal aus dem Adressraum 0x000c1ff0 bis 0x000c2000 zu lesen und eine Exception würde ausgelöst. Dies gilt es natürlich zu vermeiden.

Wir arbeiten also nur mit Adressen, die auf einer 16-Byte-Grenze liegen, was uns in Kombination mit dem Wissen, dass die Seitengröße des Betriebssystems normalerweise ebenfalls so angeordnet ist, eine gewisse Sicherheit daür bietet, dass unsere Funktion nicht aus dem nicht reservierten Speicher zu lesen versucht.

Zurück zu unserer Funktion.

`_mm_setzero_si128()`—ist ein Makro, das `pxor xmm0, xmm0` erzeugt —es löscht das XMM0 Register.

[161]wikipedia

`_mm_load_si128()`—ist ein Makro für MOVDQA: es lädt 16 Bytes von der Adresse in das XMM1 Register.

`_mm_cmpeq_epi8()`—ist ein Makro für PCMPEQB, ein Befehl, der zwei XMM-Register byteweise miteinader vergleicht.

Wenn ein Byte das gleiche ist wie im anderen Register, wird im Ergebnis hier `0xff` stehen, ansonsten aber 0.

Ein Beispiel:

```
XMM1: 0x11223344556677880000000000000000
XMM0: 0x11ab3444007877881111111111111111
```

Nach der Ausführung von `pcmpeqb xmm1, xmm0`, enthält das XMM1 Register:

```
XMM1: 0xff0000ff0000ffff0000000000000000
```

In unserem Fall vergleicht der Befehl jeden 16-Byte-Block mit einem Block aus 16 Nullbytes, der durch `pxor xmm0, xmm0` im XMM0 Register angelegt wurde.

Das nächste Makro ist `_mm_movemask_epi8()` —das ist der Befehl PMOVMSKB.

Es ist sehr nützlich im Zusammenspiel mit PCMPEQB.

`pmovmskb eax, xmm1` Dieser Befehl setzt das erste Bit in EAX auf 1, falls das MSB des ersten Bytes in XMM1 gleich 1 ist. Mit anderen Worten: wenn das erste Byte im XMM1 Register `0xff` ist, dann wird das erste Bit in EAX ebenfalls auf 1 gesetzt.

Wenn das zweite Byte in XMM1 `0xff` ist, wird das zweite Bit in EAX auf 1 gesetzt. Mit anderen Worten: der Befehl beantwortet die Frage welche Bytes in XMM1 gleich `0xff` sind und gibt 16 Bits im EAX Register zurück. Die anderen Bits im in EAX werden gelöscht.

Vergessen wir aber nicht die Eigenart in unserem Algorithmus. Es könnte im Input 16 Byte wie die folgenden geben:

15	14	13	12	11	10	9 … 3	2	1 … 0
'h'	'e'	'l'	'l'	'o'	0	Müll	0	Müll

Das ist der String `'hello'`, eine abschließende Null und ein paar Zufallswerte aus dem Speicher.

Wenn wir diese 16 Byte nach XMM1 laden und mit dem genullten XMM0 vergleichen, erhalten wir als Ergebnis etwas, das so ähnlich ist wie der folgende Output[162]:

```
XMM1: 0x0000ff00000000000000ff0000000000
```

Das bedeutet, dass der Befehl zwei Nullbytes gefunden hat. Dieses Ergebnis war zu erwarten.

PMOVMSKB in our case will set EAX to
0b0010000000100000. Offensichtlich muss unsere Funktion nur das erste Nullbit nehmen und den Rest ignorieren.

Der nächste Befehl ist BSF (*Bit Scan Forward*). Dieser Befehl erkennt, dass das erste Bit auf 1 gesetzt ist und speichert dessen Position im ersten Operanden.

[162] hier wird eine Ordnung vom **MSB!** zum **LSB!**[163] verwendet

```
EAX=0b0010000000100000
```

Nach der Ausführung von `bsf eax, eax` enthält EAX 5, was bedeutet, dass 1 an der 5. Stelle (von 0 aus gezählt) gefunden wurde.

MSVC verfügt über ein Makro für diesen Befehl: `_BitScanForward`.

Der Rest ist einfach. Wenn ein Nullbyte gefunden wurde, wird dessen Position zum Zähler hinzuaddiert und wir erhalten den Rückgabewert.

Das ist fast alles.

Hier ist bemerkenswert, dass der MSVC Compiler zur Optimierung zwei Schleifenrümpfe nebeneinander erzeugt hat.

SSE 4.2 (erschienen im Intel Core i7) verfügt über noch mehr Befehle, sodass diese Stringmanipulationen noch einfacher bewerkstelligt werden können: http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

1.28 64 Bit

1.28.1 x86-64

Es handelt sich um eine 64-Bit-Erweiterung der x86 Architektur.

Aus Sicht eines Reverse Engineers sind die wichtigsten Veränderungen die folgenden:

- Fast alle Regsiter (außer FPU und SIMD) wurden auf 64 Bit erweitert und erhielten den Präfix R-. 8 zusätzliche Register wurden hinzugefügt. Die GPR sind jetzt: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15. Es ist immernoch mgölich die älteren Registerteile wie gewohnt anzusprechen. Zum Beispiel ist es möglich, auf den niederen 32-Bit-Teil von RAX mit EAX zuzugreifen:

Byte-Nummer:							
7	6	5	4	3	2	1	0
RAX[x64]							
				EAX			
						AX	
						AH	AL

  Die neuen R8-R15 Register besitzen ebenfalls niedere Teile: R8D-R15D (niedere 32 Bit), R8W-R15W (niedere 16 Bit), R8L-R15L (niedere 8 Bit).

Byte-Nummer:							
7	6	5	4	3	2	1	0
R8							
				R8D			
						R8W	
							R8L

  Die Anzahl der SIMD Register wurde von 8 auf 16 erhöht:XMM0-XMM15.

- In Win64 ist die Konvention zum Aufruf von Funktionen ein wenig verändert worden und erinnert nun an fastcall(6.1.3 on page 582). Die ersten vier Argumente werden in den RCX, RDX, R8 und R9 Registern gespeichert —der Rest auf dem Stack. Die aufrufende Funktion muss also 32 Byte reservieren, sodass der Aufgerufene die ersten 4 Argumente dort speichern und sie für eigene Zwecke verwenden kann. Kurze Funktionen können Argumente verwenden, die ausschließlich aus Registern stammen, aber größere müssen ihre Variablen auf dem Stack speichern.

  System V AMD64 ABI (Linux, *BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [164]erinnert auch an fastcall: es verwendet die 6 Register RDI, RSI, RDX, RCX, R8, R9 für die ersten 6 Argumente. Der Rest wird wie gehabt über den Stack übergeben.

  Siehe dazu auch den Abschnitt über Aufrufkonventionen (6.1 on page 580).

- Der Typ *int* in C/C++ hat aus Kompatibilitätsgründen immernoch die Größe 32 Bit.

- Alle Pointer haben jetzt eine Größe von 64 Bit.

Da sich die Anzahl der Register verdoppelt hat, hat der Compiler mehr Spielraum für den . Für uns hat dies zur Folge, dass der erzeugte Code eine geringere Anzahl lokaler Variabeln enthält.

Die Funktionm, die die erste S-Box im DES Verschlüsselungsalgorithmus berechnet, verarbeitet beispielsweise 32/64/128/256 Werte auf einmal (abhängig vom DES_type: uint32, uint64, SSE2 oder AVX)) mithilfe der Bitslice DES Methode (mehr zu dieser Technik unter (1.27 on page 483)):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
```

[164]Auch verfügbar als https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
```

```
    x37 = x18 & ~x36;
    x38 = a2 | x3;
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

Es gibt eine Menge lokaler Variablen. Natürlich werden diese nicht alle auf dem lokalen Stack gespeichert. Kompilieren wir mit MSVC 2008 mit der Option 0x:

Listing 1.368: Optimierender MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
_a6$ = 28           ; size = 4
_out1$ = 32         ; size = 4
_x24$ = 36          ; size = 4
_out2$ = 36         ; size = 4
_out3$ = 40         ; size = 4
_out4$ = 44         ; size = 4
_s1    PROC
```

```
    sub     esp, 20                         ; 00000014H
    mov     edx, DWORD PTR _a5$[esp+16]
    push    ebx
    mov     ebx, DWORD PTR _a4$[esp+20]
    push    ebp
    push    esi
    mov     esi, DWORD PTR _a3$[esp+28]
    push    edi
    mov     edi, ebx
    not     edi
    mov     ebp, edi
    and     edi, DWORD PTR _a5$[esp+32]
    mov     ecx, edx
    not     ecx
    and     ebp, esi
    mov     eax, ecx
    and     eax, esi
    and     ecx, ebx
    mov     DWORD PTR _x1$[esp+36], eax
    xor     eax, ebx
    mov     esi, ebp
    or      esi, edx
    mov     DWORD PTR _x4$[esp+36], esi
    and     esi, DWORD PTR _a6$[esp+32]
    mov     DWORD PTR _x7$[esp+32], ecx
    mov     edx, esi
    xor     edx, eax
    mov     DWORD PTR _x6$[esp+36], edx
    mov     edx, DWORD PTR _a3$[esp+32]
    xor     edx, ebx
    mov     ebx, esi
    xor     ebx, DWORD PTR _a5$[esp+32]
    mov     DWORD PTR _x8$[esp+36], edx
    and     ebx, edx
    mov     ecx, edx
    mov     edx, ebx
    xor     edx, ebp
    or      edx, DWORD PTR _a6$[esp+32]
    not     ecx
    and     ecx, DWORD PTR _a6$[esp+32]
    xor     edx, edi
    mov     edi, edx
    or      edi, DWORD PTR _a2$[esp+32]
    mov     DWORD PTR _x3$[esp+36], ebp
    mov     ebp, DWORD PTR _a2$[esp+32]
    xor     edi, ebx
    and     edi, DWORD PTR _a1$[esp+32]
    mov     ebx, ecx
    xor     ebx, DWORD PTR _x7$[esp+32]
    not     edi
    or      ebx, ebp
    xor     edi, ebx
    mov     ebx, edi
```

```
    mov     edi, DWORD PTR _out2$[esp+32]
    xor     ebx, DWORD PTR [edi]
    not     eax
    xor     ebx, DWORD PTR _x6$[esp+36]
    and     eax, edx
    mov     DWORD PTR [edi], ebx
    mov     ebx, DWORD PTR _x7$[esp+32]
    or      ebx, DWORD PTR _x6$[esp+36]
    mov     edi, esi
    or      edi, DWORD PTR _x1$[esp+36]
    mov     DWORD PTR _x28$[esp+32], ebx
    xor     edi, DWORD PTR _x8$[esp+36]
    mov     DWORD PTR _x24$[esp+32], edi
    xor     edi, ecx
    not     edi
    and     edi, edx
    mov     ebx, edi
    and     ebx, ebp
    xor     ebx, DWORD PTR _x28$[esp+32]
    xor     ebx, eax
    not     eax
    mov     DWORD PTR _x33$[esp+32], ebx
    and     ebx, DWORD PTR _a1$[esp+32]
    and     eax, ebp
    xor     eax, ebx
    mov     ebx, DWORD PTR _out4$[esp+32]
    xor     eax, DWORD PTR [ebx]
    xor     eax, DWORD PTR _x24$[esp+32]
    mov     DWORD PTR [ebx], eax
    mov     eax, DWORD PTR _x28$[esp+32]
    and     eax, DWORD PTR _a3$[esp+32]
    mov     ebx, DWORD PTR _x3$[esp+36]
    or      edi, DWORD PTR _a3$[esp+32]
    mov     DWORD PTR _x36$[esp+32], eax
    not     eax
    and     eax, edx
    or      ebx, ebp
    xor     ebx, eax
    not     eax
    and     eax, DWORD PTR _x24$[esp+32]
    not     ebp
    or      eax, DWORD PTR _x3$[esp+36]
    not     esi
    and     ebp, eax
    or      eax, edx
    xor     eax, DWORD PTR _a5$[esp+32]
    mov     edx, DWORD PTR _x36$[esp+32]
    xor     edx, DWORD PTR _x4$[esp+36]
    xor     ebp, edi
    mov     edi, DWORD PTR _out1$[esp+32]
    not     eax
    and     eax, DWORD PTR _a2$[esp+32]
    not     ebp
```

```
    and     ebp, DWORD PTR _a1$[esp+32]
    and     edx, esi
    xor     eax, edx
    or      eax, DWORD PTR _a1$[esp+32]
    not     ebp
    xor     ebp, DWORD PTR [edi]
    not     ecx
    and     ecx, DWORD PTR _x33$[esp+32]
    xor     ebp, ebx
    not     eax
    mov     DWORD PTR [edi], ebp
    xor     eax, ecx
    mov     ecx, DWORD PTR _out3$[esp+32]
    xor     eax, DWORD PTR [ecx]
    pop     edi
    pop     esi
    xor     eax, ebx
    pop     ebp
    mov     DWORD PTR [ecx], eax
    pop     ebx
    add     esp, 20
    ret     0
_s1    ENDP
```

Der Compiler legt auf dem lokalen Stack 5 Variablen an.

Untersuchen wir dasselbe Programm in der 64-Bit-Version mit MSVC 2008:

Listing 1.369: Optimierender MSVC 2008

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    mov     r15, QWORD PTR a5$[rsp]
```

```
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not     rax
    xor     rdx, r9
    not     r10
    mov     r11, rax
    and     rax, r9
    mov     rsi, r10
    mov     QWORD PTR x36$1$[rsp], rax
    and     r11, r8
    and     rsi, r8
    and     r10, r15
    mov     r13, rdx
    mov     rbx, r11
    xor     rbx, r9
    mov     r9, QWORD PTR a2$[rsp]
    mov     r12, rsi
    or      r12, r15
    not     r13
    and     r13, rcx
    mov     r14, r12
    and     r14, rcx
    mov     rax, r14
    mov     r8, r14
    xor     r8, rbx
    xor     rax, r15
    not     rbx
    and     rax, rdx
    mov     rdi, rax
    xor     rdi, rsi
    or      rdi, rcx
    xor     rdi, r10
    and     rbx, rdi
    mov     rcx, rdi
    or      rcx, r9
    xor     rcx, rax
    mov     rax, r13
    xor     rax, QWORD PTR x36$1$[rsp]
    and     rcx, QWORD PTR a1$[rsp]
    or      rax, r9
    not     rcx
    xor     rcx, rax
    mov     rax, QWORD PTR out2$[rsp]
    xor     rcx, QWORD PTR [rax]
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR x36$1$[rsp]
    mov     rcx, r14
    or      rax, r8
    or      rcx, r11
```

```
    mov     r11, r9
    xor     rcx, rdx
    mov     QWORD PTR x36$1$[rsp], rax
    mov     r8, rsi
    mov     rdx, rcx
    xor     rdx, r13
    not     rdx
    and     rdx, rdi
    mov     r10, rdx
    and     r10, r9
    xor     r10, rax
    xor     r10, rbx
    not     rbx
    and     rbx, r9
    mov     rax, r10
    and     rax, QWORD PTR a1$[rsp]
    xor     rbx, rax
    mov     rax, QWORD PTR out4$[rsp]
    xor     rbx, QWORD PTR [rax]
    xor     rbx, rcx
    mov     QWORD PTR [rax], rbx
    mov     rbx, QWORD PTR x36$1$[rsp]
    and     rbx, rbp
    mov     r9, rbx
    not     r9
    and     r9, rdi
    or      r8, r11
    mov     rax, QWORD PTR out1$[rsp]
    xor     r8, r9
    not     r9
    and     r9, rcx
    or      rdx, rbp
    mov     rbp, QWORD PTR [rsp+80]
    or      r9, rsi
    xor     rbx, r12
    mov     rcx, r11
    not     rcx
    not     r14
    not     r13
    and     rcx, r9
    or      r9, rdi
    and     rbx, r14
    xor     r9, r15
    xor     rcx, rdx
    mov     rdx, QWORD PTR a1$[rsp]
    not     r9
    not     rcx
    and     r13, r10
    and     r9, r11
    and     rcx, rdx
    xor     r9, rbx
    mov     rbx, QWORD PTR [rsp+72]
    not     rcx
```

```
    xor     rcx, QWORD PTR [rax]
    or      r9, rdx
    not     r9
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR out3$[rsp]
    xor     r9, r13
    xor     r9, QWORD PTR [rax]
    xor     r9, r8
    mov     QWORD PTR [rax], r9
    pop     r15
    pop     r14
    pop     r13
    pop     r12
    pop     rdi
    pop     rsi
    ret     0
s1    ENDP
```

Hier wurde im lokalen Stack nichts durch den Compiler angelegt, x36 ist ein Synonym für a5.

Es gibt übrigens CPUs mit viel mehr GPRs, z.B. Itanium (128 Register).

1.28.2 ARM

64-Bit-Befehle erschienen in ARMv8.

1.28.3 Fließkommazahlen

Den Umgang mit Fließkommazahlen in x86-64 haben wir hier erklärt: 1.29 on the following page.

1.28.4 Kritik an der 64-Bit-Architektur

Einige Leute sind manchmal am Zweifeln bezüglich der Vorteil von 64-Bit-Architekturen: man benötigt nun doppelt soviel Platz für das Speichern von Pointern, inklusive Cachespeicher, obwohl die x64 CPUs nur 48 Bit an externem RAM adressieren können.

Einige leute schrieben ihre eigenen Speicherreservierungen.

Interessant ist dabei der Fall CryptoMiniSat[165]. Dieses Programm benötigt fast nie mehr als 4GiB RAM, macht aber starken Gebrauch von Pointern. Es benötigt also auf einer 32-Bit-Architektur weniger Speicher als unter einer 64-Bit-Architektur. Um dieses Problem zu lösen, hat der Autor seine eigene Speicherreservierung (in den Dateien *clauseallocator.(h|cpp)*) geschrieben, welche erlaubt, Speicher mit 32-Bit-Identifiern anstelle von 64-Bit-Pointern zu reservieren.

[165] https://github.com/msoos/cryptominisat/

1.29 Arbeiten mit Fließkommazahlen und SIMD

Natürlich verblieb die FPU in x86-kompatiblen Prozessoren als die SIMD Erweiterungen hinzugefügt wurden.

Die SIMD Erweiterungen (SSE2) bieten einen einfacheren Weg um mit Fließkommazahlen zu arbeiten.

Das Zahlenformat bleibt dabei das gleich (IEEE 754).

Moderne Compiler (inklusive derer für x86-64) verwenden normalerweise SIMD Befehle anstatt FPU-Befehle.

Wir werden hier die Beispiele aus dem Abschnitt über die FPU recyclen: 1.19 on page 253.

1.29.1 Ein einfaches Beispiel

```
#include <stdio.h>

double f (double a, double b)
{
        return a/3.14 + b*4.1;
};

int main()
{
        printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.370: Optimierender MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Die eingegebenen Fließkommawerte werden in die Register XMM0-XMM3 übergeben und der Rest über den StackfootnoteMSDN: Parameterübergabe.

a wird nach XMM0 übergeben und b nach XMM1.

Die XMM Register sind 128 Bit breit (wie wir bereits aus dem Abschnitt über SIMD wissen: 1.27 on page 482), aber die *double* Werte umfassen nur 64 Bit, sodass nur die niedere Hälfte der Register benötigt wird.

DIVSD ist ein SSE-Befehl, der für „Divide Scalar Double-Precision Floating-Point Values“ steht; er teilt einen *double* Wert durch einen anderen, wobei beide in den niederen Hälften der Operanden gespeichert sind.

Die Konstanten werden durch den Compiler im IEEE 754 Format kodiert.

MULSD und ADDSD funktionieren genauso und führen Multiplikation und Addition durch.

Das Ergebnis der Funktionsausführung ist von Typ *double* und wird im XMM0 Register abgelegt.

So arbeitet der nicht optimierende MSVC:

Listing 1.371: MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Ein wenig redundant. Die Eingabewerte bzw. deren niedere Registerhälften, d.h. die 64-Bit-Werte vom Typ *double*, werden im „shadow space“(1.10.2 on page 113) gespeichert. GCC erzeugt identischen Code.

x86

Kompilieren wir das Beispiel für x86. Obwohl der Code für x86 erzeugt wird, verwendet MSVC 2012 SSE2 Befehle:

Listing 1.372: Nicht optimierender MSVC 2012 x86

```
tv70 = -8       ; size = 8
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f      PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        movsd   xmm0, QWORD PTR _a$[ebp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
```

```
        movsd   xmm1, QWORD PTR _b$[ebp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        movsd   QWORD PTR tv70[ebp], xmm0
        fld     QWORD PTR tv70[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_f      ENDP
```

Listing 1.373: Optimierender MSVC 2012 x86

```
tv67 = 8        ; size = 8
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f      PROC
        movsd   xmm1, QWORD PTR _a$[esp-4]
        divsd   xmm1, QWORD PTR __real@40091eb851eb851f
        movsd   xmm0, QWORD PTR _b$[esp-4]
        mulsd   xmm0, QWORD PTR __real@4010666666666666
        addsd   xmm1, xmm0
        movsd   QWORD PTR tv67[esp-4], xmm1
        fld     QWORD PTR tv67[esp-4]
        ret     0
_f      ENDP
```

Es ist fast der gleiche Code, aber es gibt einige Unterschiede in Bezug auf die Aufrufkonventionen: 1) die Argumente werden nicht über die XMM Register, sondern über den Stack übergeben, wie in den FPU Beispielen (1.19 on page 253); 2) das Ergebnis der Funktion wird über `ST(0)` zurückgegeben—um dies zu erreichen wird es (durch die lokale Variable `tv`) aus einem der XMM Register nach `ST(0)` kopiert.

Untersuchen wir das optimierte Beispiel mit OllyDbg:

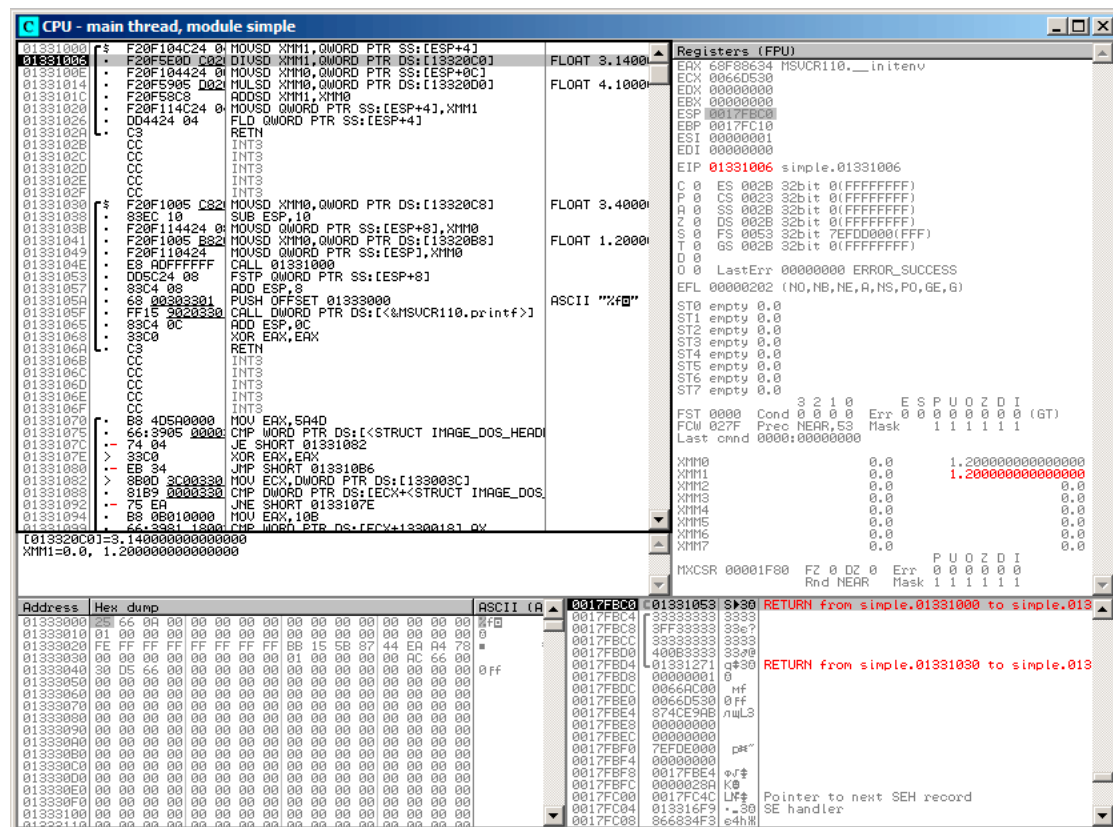


Abbildung 1.113: OllyDbg: MOVSD lädt den Wert von a nach XMM1

Abbildung 1.114: OllyDbg: DIVSD hat den Quotienten berechnet und in XMM1 gespeichert

Abbildung 1.115: OllyDbg: MULSD hat das Produkt berechnet und in XMM0 gespeichert

Abbildung 1.116: OllyDbg: ADDSD addiert den Wert in XMM0 zu XMM1

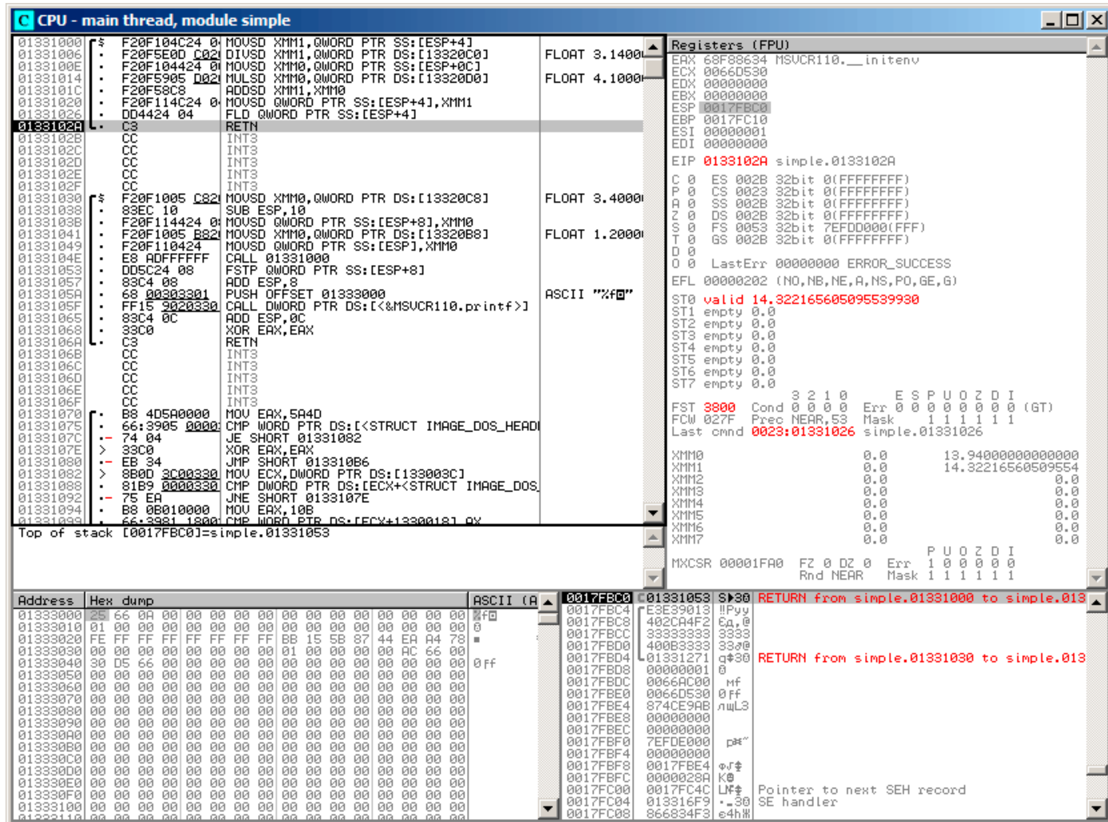


Abbildung 1.117: OllyDbg: FLD lässt das Funktionsergebnis in ST(0)

Wir sehen, dass OllyDbg die XMM Register als Paare von *double* Zahlen anzeigt, aber nur der niedere Teil davon verwendet wird.

Offenbar zeigt OllyDbg sie in diesem Format an, weil die SSE2 Befehle (die mit dem Suffix -SD) jetzt ausgeführt werden.

Natürlich ist es auch möglich das Registerformat zu ändern und sich die Inhalte als 4 *float*-Zahlen oder nur als 16 Byte anzeigen zu lassen.

1.29.2 Fließkommazahlen als Argumente übergeben

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

Sie werden über die niederen Hälften der Register XMM0-XMM3 übergeben.

Listing 1.374: Optimierender MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r   ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r   ; 1.54

main    PROC
        sub     rsp, 40                                 ; 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                 ; 00000028H
        ret     0
main    ENDP
```

Es gibt in Intel keinen MOVSDX Befehl und AMD-Handbücher (11.1.4 on page 677) bezeichnen ihn nur mit MOVSD. Es gibt insgesamt zwei Befehle, die sich in x86 denselben Namen teilen (für den anderen siehe: **??** on page ??). Offenbar wollten die Microsoft Entwickler hier aufräumen und haben ihn deshalb in MOVSDX umbenannt. Er lädt lediglich einen Wert in die niedere Hälfte eines XMM Registers.

pow() nimmt Argumente aus XMM0 und XMM1 und gibt das Ergebnis in XMM0 zurück. Es wird dann für printf() nach RDX verschoben. Der Grund dafür ist möglicherweise, dass printf() eine Funktion mit einer variablen Anzahl an Argumenten ist.

Listing 1.375: Optimierender GCC 4.4.6 x64

```
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
main:
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
        call    pow
```

```
        ; result is now in XMM0
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 1 ; Anzahl der übergebenen Vektorregister
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
.LC0:
        .long   171798692
        .long   1073259479
.LC1:
        .long   2920577761
        .long   1077936455
```

GCC erzeugt klarer strukturierten Output. Der Wert für `printf()` wird in `XMM0` übergeben. Hier ist übrigens ein Fall, in dem 1 nach EAX für `printf()` geschrieben wird um anzuzeigen, dass ein Argument in den Vektorregistern übergeben wird, genau wie es der Standard [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] [166]verlangt.

1.29.3 Beispiel mit Vergleich

```
#include <stdio.h>

double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};

int main()
{
        printf ("%f\n", d_max (1.2, 3.4));
        printf ("%f\n", d_max (5.6, -4));
};
```

x64

Listing 1.376: Optimierender MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        comisd  xmm0, xmm1
        ja      SHORT $LN2@d_max
```

[166] Auch verfügbar als https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

```
        movaps  xmm0, xmm1
$LN2@d_max:
        fatret  0
d_max   ENDP
```

Optimierender MSVC erzeugt leicht verständlichen Code.

COMISD bedeutet „Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS“. Das beschreibt ziemlich genau, was der Befehl tatsächlich tut.

Nicht optimierender MSVC erzeugt Code mit mehr Redundanzen, der aber immer noch gut verständlich ist:

Listing 1.377: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max:
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret  0
d_max   ENDP
```

GCC 4.4.6 hat mehr Optimierungen durchgeführt und den Befehl MAXSD („Return Maximum Scalar Double-Precision Floating-Point Value“) verwendet, der einfach den größten Wert auswählt!

Listing 1.378: Optimierender GCC 4.4.6 x64

```
d_max:
        maxsd   xmm0, xmm1
        ret
```

x86

Kompilieren wir dieses Beispiel in MSVC 2012 mit aktivierter Optimierung:

Listing 1.379: Optimierender MSVC 2012 x86

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_d_max  PROC
        movsd   xmm0, QWORD PTR _a$[esp-4]
        comisd  xmm0, QWORD PTR _b$[esp-4]
        jbe     SHORT $LN1@d_max
        fld     QWORD PTR _a$[esp-4]
        ret     0
$LN1@d_max:
        fld     QWORD PTR _b$[esp-4]
        ret     0
_d_max  ENDP
```

Fast identisch, nur dass die Werte a und b vom Stack geholt werden und das Funktionsergebnis in `ST(0)` gelassen wird.

Wenn wir dieses Beispiel in OllyDbg laden, erkennen wir, wie der Befehl `COMISD` Werte vergleicht und die `CF` und `PF` Flags setzt bzw. löscht:

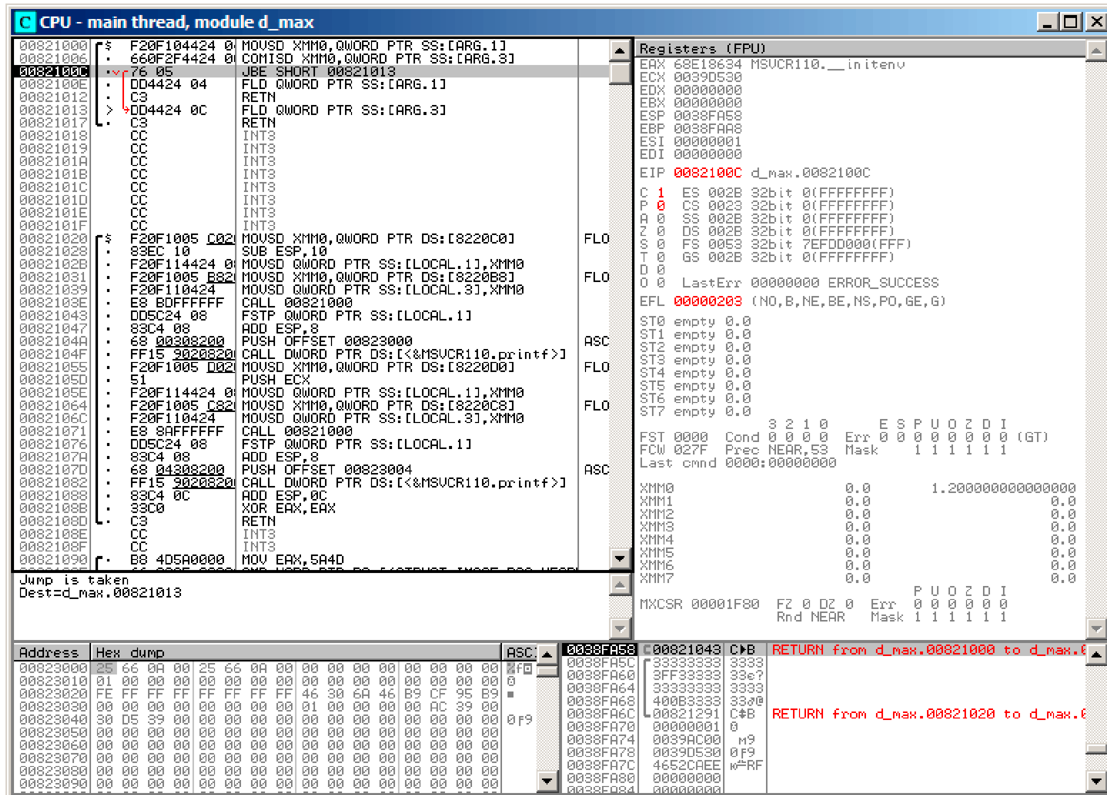


Abbildung 1.118: OllyDbg: COMISD hat die CF und PF Flags verändert

1.29.4 Berechnen der Maschinengenauigkeit: x64 und SIMD

Betrachten wir erneut das Beispiel zur Berechnung der Maschinengenauigkeit für *double* Listing.1.24.2.

Wir kompilieren es jetzt für x64:

Listing 1.380: Optimierender MSVC 2012 x64

```
v$ = 8
calculate_machine_epsilon PROC
        movsdx   QWORD PTR v$[rsp], xmm0
        movaps   xmm1, xmm0
        inc      QWORD PTR v$[rsp]
        movsdx   xmm0, QWORD PTR v$[rsp]
        subsd    xmm0, xmm1
        ret      0
calculate_machine_epsilon ENDP
```

Es gibt keinen Weg 1 zum einem Wert in einem 128-Bit XMM Register zu addieren, also muss der Wert im Speicher abgelegt werden.

Es gibt zwar den Befehl ADDSD (*Add Scalar Double-Precision Floating-Point Values*), der einen Wert zu niederen 64-Bit-Hälfte eines XMM Registers addieren kann und die höheren Bits ignoriert, aber MSVC 2012 scheint an dieser Stelle nicht gut genug zu sein, um diese Möglichkeit zu erkennen[167].

Nichtsdestotrotz wird der Wert dann wieder in ein XMM Register geladen und eine Subtraktion wird durchgeführt. SUBSD steht für „Subtract Scalar Double-Precision Floating-Point Values“, d.h. er arbeitet nur auf dem niederen 64-Bit-Teil des 128-Bit XMM Registers. Das Ergebnis wird in das XMM0-Register zurückgegeben.

1.29.5 Erneute Betrachtung des Beispiels zum Pseudozufallszahlengenerator

Betrachten wir erneut das Beispiel zum Pseudozufallszahlengenerator Listing.1.24.1.

Wenn wir es in MSVC 2012 kompilieren, werden SIMD Befehle für die FPU benutzt.

Listing 1.381: Optimierender MSVC 2012

```
__real@3f800000 DD 03f800000r   ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
        push    ecx
        call    ?my_rand@@YAIXZ
; EAX=Pseudozufallswert
        and     eax, 8388607    ; 007fffffH
        or      eax, 1065353216 ; 3f800000H
; EAX=Pseudozufallswert & 0x007fffff | 0x3f800000
; speichere ihn auf lokalem Stack:
        mov     DWORD PTR _tmp$[esp+4], eax
; lade ihn erneut als Fließkommazahl:
        movss   xmm0, DWORD PTR _tmp$[esp+4]
; subtrahiere 1.0:
        subss   xmm0, DWORD PTR __real@3f800000
; verschiebe Wert nach ST0 durch Ablegen in temporärer Variable…
        movss   DWORD PTR tv128[esp+4], xmm0
; ... und lade ihn erneut nach ST0:
        fld     DWORD PTR tv128[esp+4]
        pop     ecx
        ret     0
?float_rand@@YAMXZ ENDP
```

Alle Befehle haben den Suffix -SS, der für „Scalar Single“ steht.

„Scalar“ bedeutet, dass nur ein Wert im Register gespeichert ist. „Single“[168] steht für den Datentyp *float*.

[167]Als Übung können Sie versuchen, den Code so umzugestalten, dass der lokale Stack nicht mehr verwendet wird
[168]Single precision

1.29.6 Zusammenfassung

In den Beispielen hier wird nur die niedere Hälfte der XMM Register verwendet, um eine Zahl im IEEE 754 Format zu speichern.

Im Prinzip arbeiten alle Befehle, die um den Präfix `-SD` („Scalar Double-Precision") ergänzt wurden, mit Fließkommazahlen im IEEE 754 Format, die in der niederen 64-Bit-Hälfte eines XMM Registers gespeichert werden.

Dies ist einfacher als in der FPU, da die SIMD sich in weniger chaotischer Weise als die FPU entwickelt hat. Das Stackregister Modell wird nicht verwendet.

Wenn man in diesen Beispielen *double* durch *float* ersetzen würde, würden die gleichen Befehle verwendet werden, aber jeweils mit `-SS`(„Scalar Single-Precision") Präfix, z.B. `MOVSS`, `COMISS`, `ADDSS`, etc.

„Scalar" bedeutet, dass die SIMD Register nur einen anstatt mehreren Werten enthalten.

Befehle, die mit mehreren Werte in einem Register gleichzeitig arbeiten, haben ein „Packed" in ihrem Namen.

Unnötig extra zu erwähnen, dass die SSE2 Befehle mit 64-Bit IEEE 743 Werten (*double*) arbeiten, wohingegen die interne Repräsentation von Fließkommazahlen in der FPU 80-Bit-Zahlen verwendet.

Die FPU wird deshalb manchmal weniger Rundungsfehler machen und als Konsequenz daraus möglicherweise präzisere Berechnungsergebnisse liefern.

1.30 ARM-spezifische Details

1.30.1 Zeichen (#) vor einer Zahl

Der Keil-Compiler, IDA und objdump versehen alle Zahlen mit dem Präfix „#", siehe hier: Listing.1.16.1.

Wenn GCC 4.9 Output in Assemblersprache erzeugt, tut er dies jedoch ohne den Präfix: Listing.**??**.

Die ARM-Listings in diesem Buch sind gemischt.

Es ist schwer zu sagen, welche Methode die richtige ist. Am einfachsten ist es, die Regeln, die in der Umgebung, mit der man arbeitet, vorherrschen, zu akzeptieren.

1.30.2 Adressierungsmodi

Der folgende Befehl ist in ARM64 zulässig:

```
ldr     x0, [x29,24]
```

Das bedeutet, wir addieren 24 zum Wert in X29 und laden den Wert an dieser Adresse

Man beachte, dass die 24 sich innerhalb der Klammern befindet. Die Bedeutung ist eine andere, wenn die Zahl außerhalb der Klammer steht:

```
ldr     w4, [x1],28
```

Das bedeutet, wir laden den Wert an der Adresse in X1 und addieren dann 28 zu X1.

ARM erlaubt die Addition oder Subtraktion einer Konstanten zu bzw. von einer für einen Ladebefehl benötigten Adresse.

Es ist möglich dies vor und nach dem Laden zu tun.

Es gibt keinen derartigen Adressierungsmodus in x86, aber auch in anderen Prozessoren, sogar auf PDP-11.

Es gibt die Legende, dass die Pre-Inkrement-, Post-Inkrement-, Pre-Dekrement und Post-Dekrement-Modi in PDP-11 für das Erscheinen von C-Konstrukten (welche auf PDP-11 entwickelt wurden) wie *ptr++, *++ptr, *ptr--, *--ptr verantwortlich sind.

Dabei handelt es sich übrigens um ein schwer zu merkendes Feature von C. Es funktioniert wie folgt:

C Term	ARM Term	C Ausdruck	so funktioniert er
Post-Inkrement	post-indexed Adressierung	`*ptr++`	verwende `*ptr` Wert, dann inkrementiere `ptr` Pointer
Post-Dekrement	post-indexed Adressierung	`*ptr--`	verwende `*ptr` Wert, dann dekrementiere `ptr` Pointer
Prä-Inkrement	pre-indexed Adressierung	`*++ptr`	inkrementiere `ptr` Pointer, dann verwende `*ptr` Wert
Prä-Dekrement	pre-indexed Adressierung	`*--ptr`	dekrementiere `ptr` Pointer, dann verwende `*ptr` Wert

Pre-indexing wird in der ARM Assemblersprache mit einem Ausrufezeichen kenntlich gemacht. Zum Beispiel in der Zeile 2 in Listing.1.27.

Dennis Ritchie (einer der Erfinder von C) hat erwähnt, dass diese Modi vermutlich deshalb von Ken Thompson (einem anderen Erfinder von C) erfunden wurde, weil es dieses Prozessorfeature in PDP-7 bereits gab[169], [Dennis M. Ritchie, *The development of the C language*, (1993)][170].

Dadurch können C-Compiler das Feature nutzen, wenn es auch auf dem Zielprozessor implementiert ist.

Es ist sehr nützlich und gebräuchlich beim Verarbeiten von Arrays.

[169] `http://yurichev.com/mirrors/C/c_dmr_postincrement.txt`
[170] Auch verfügbar als pdf

1.30.3 Laden einer Konstante in ein Register

32-Bit ARM

Wie wir bereits wissen, haben alle Befehle im ARM mode eine Länge von 4 Byte bzw. 2 Byte im Thumb mode.

Wie können wir nun einen 32-Bit-Wert in ein Register laden, wenn es nicht möglich ist, ihn in einen Befehl hineinzukodieren?

Versuchen wir es:

```
unsigned int f()
{
        return 0x12345678;
};
```

Listing 1.382: GCC 4.6.3 -O3 ARM Modus

```
f:
        ldr     r0, .L2
        bx      lr
.L2:
        .word   305419896 ; 0x12345678
```

Der Wert `0x12345678` wird im Speicher geparkt und wenn nötig geladen. Es ist aber möglich, den zusätzlichen Speicherzugriff loszuwerden.

Listing 1.383: GCC 4.6.3 -O3 -march=armv7-a (ARM Modus)

```
movw    r0, #22136      ; 0x5678
movt    r0, #4660       ; 0x1234
bx      lr
```

Wir sehen, dass der Wert stückweise in das Register geladen wird: zuerst der niedere Teil (mit `MOVW`), dann der höhere (mit `MOVT`).

Das bedeutet, dass im ARM mode 2 Befehle nötig sind, um einen 32-Bit-Wert in ein Register zu laden.

Das stellt kein wirkliches Problem dar, denn es gibt in realen Code nicht allzu viele Konstanten (außer 0 und 1).

Hat dies auch zur Folge, dass die Version mit zwei Befehlen langsamer ist als die mit einem?

Eher nicht. Wahrscheinlicher ist, dass moderne ARM Prozessoren in der Lage sind, solche Sequenzen zu erkennen und schnell auszuführen.

Auch IDA vermag solche Muster im Code zu erkennen und disassembliert die Funktion wie folgt:

```
MOV     R0, 0x12345678
BX      LR
```

ARM64

```
uint64_t f()
{
        return 0x12345678ABCDEF01;
};
```

Listing 1.384: GCC 4.9.1 -O3

```
mov     x0, 61185   ; 0xef01
movk    x0, 0xabcd, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
ret
```

MOVK steht für „MOV Keep“, d.h. der Befehl schreibt einen 16-Bit-Wert in das Register und lässt die übrigen Bits unverändert. Der Suffix LSL verschiebt den Wert um 16, 32 und 48 Bits in jedem Schritt nach links. Das Verschieben wird vor dem Laden durchgeführt.

Das bedeutet, dass 4 Befehle notwendig sind, um einen 64-Bit-Wert in ein Register zu laden.

Fließkommazahl in einem Register speichern

Es ist möglich mit nur einem Befehl eine Fließkommazahl in einem D-Register zu speichern.

Ein Beispiel:

```
double a()
{
        return 1.5;
};
```

Listing 1.385: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
   0:   1e6f1000        fmov    d0, #1.500000000000000000e+000
   4:   d65f03c0        ret
```

Die Zahl 1.5 war tatsächlich in dem 32-Bit-Befehl kodiert. Aber wie ist das möglich? In ARM64 stehen 8 Bits im FMOV Befehl für das Kodieren von Fließkommazahlen zur Verfügung. Der Algorithmus heißt VFPExpandImm() in [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][171].

Dieses Vorgehen wird auch *minifloat* genannt[172].

Wir können verschiedene Werte ausprobieren: der Compiler ist in der Lage 30.0 und 31.0 zu kodieren, aber nicht 32.0, da hier 8 Bytes für diese Zahl im IEEE 754 Format reserviert werden müssen.

[171] Auch verfügbar als http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf

[172] wikipedia

```
double a()
{
        return 32;
};
```

Listing 1.386: GCC 4.9.1 -O3

```
a:
        ldr     d0, .LC0
        ret
.LC0:
        .word   0
        .word   1077936128
```

1.30.4 Relocs in ARM64

Wie wir wissen, gibt es 4-Byte-Befehle in ARM64, sodass es unmöglich ist, eine große Zahl mit einem einzigen Befehl in ein Register zu schreiben.

Nichtsdestotrotz kann eine Executable an jeder Adresse des Speichers geladen werden und das ist der Grund dafür, dass Relocs existieren. Mehr dazu (in Bezug auf Win32 PE) unter: 6.5.2 on page 615.

Die Adresse wird in ARM64 aus einem ADRP und ADD Befehlspaar zusammengesetzt.

Der erste Befehl lädt eine 4KiB-Seitenadresse und der zweite den Rest. Kompilieren wir das Beispiel aus „Hallo, Welt!“ (Listing.1.8) in GCC (Linaro) 4.9 unter win32:

Listing 1.387: GCC (Linaro) 4.9 und objdump der Objektdatei

```
...>aarch64-linux-gnu-gcc.exe hw.c -c

...>aarch64-linux-gnu-objdump.exe -d hw.o

...

0000000000000000 <main>:
   0:   a9bf7bfd        stp     x29, x30, [sp,#-16]!
   4:   910003fd        mov     x29, sp
   8:   90000000        adrp    x0, 0 <main>
   c:   91000000        add     x0, x0, #0x0
  10:   94000000        bl      0 <printf>
  14:   52800000        mov     w0, #0x0    // #0
  18:   a8c17bfd        ldp     x29, x30, [sp],#16
  1c:   d65f03c0        ret

...>aarch64-linux-gnu-objdump.exe -r hw.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21  .rodata
```

```
000000000000000c R_AARCH64_ADD_ABS_LO12_NC  .rodata
0000000000000010 R_AARCH64_CALL26  printf
```

Es gibt hier 3 Relocs in dieser Datei.

- Der erste nimmt die Seitenadresse, schneidet die niederstnen 12 Bits aus und schreibt die übrigen 21 Bit in das Bitfield für den ADRP Befehl. Das wird getan, weil wir die niedersten 12 Bit nicht kodieren und der Befehl ADRP nur Platz für 21 Bit hat.
- Der zweite legt die 12 Bit der Adresse relativ zum Seitenanfang in das Bitfield des ADD Befehls.
- Der letzte mit 26 Bit wird auf den Befehl an der Adresse `0x10` angewendet, an der sich der Sprung zur Funktion `printf()` befindet.

  Alle ARM64 (und ARM im ARM mode) Befehlsadressen haben Nullen in den zwei niederwertigsten Bits (da alle Befehle eine Größe von 4 Byte haben), sodass man nur die höchsten 26 Bit eines 28-Bit-Adressraums (± 128MB) kodieren muss.

Solche Relocs gibt es in der ausführbaren Datei nicht: das liegt daran, dass bekannt ist, wo der „Hello!" String abgelegt wurde; die Seiten und die Adresse von `puts()` sind ebenfalls bekannt.

In den Befehlen ADRP, ADD und BL sind also bereits Werte gesetzt (der Linker hat diese während des Linkens geschrieben):

Listing 1.388: objdump der ausführbaren Datei

```
0000000000400590 <main>:
  400590:       a9bf7bfd        stp     x29, x30, [sp,#-16]!
  400594:       910003fd        mov     x29, sp
  400598:       90000000        adrp    x0, 400000 <_init-0x3b8>
  40059c:       91192000        add     x0, x0, #0x648
  4005a0:       97ffffa0        bl      400420 <puts@plt>
  4005a4:       52800000        mov     w0, #0x0  // #0
  4005a8:       a8c17bfd        ldp     x29, x30, [sp],#16
  4005ac:       d65f03c0        ret

...

Contents of section .rodata:
 400640 01000200 00000000 48656c6c 6f210000  ........Hello!..
```

Zur Veranschaulichung wollen wir den BL Befehl manuell disassemblieren.
`0x97ffffa0` entspricht $0b10010111111111111111111110100000$. Gemäß [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)C5.6.26] sind *imm26* die letzten 26 Bit:
$imm26 = 0b11111111111111111110100000$. Das ist `0x3FFFFA0`, aber da das MSB 1 ist, handelt es sich um eine negative Zahl und wir können diese manuell in eine handlichere Form umwandeln. Nach den Regeln für Negation invertieren wir alle Bits: (das ergibt `0b1011111=0x5F`) und addieren dann 1 (`0x5F+1=0x60`). Die vorzeichenbehaftete Form ist also `-0x60`. Multiplizieren wir `-0x60` mit 4 (da die im Opcode gespeicherte

Adresse durch 4 geteilt worden ist): das ergibt `-0x180`. Berechnen wir nun die Zieladresse: `0x4005a0` + (`-0x180`) = `0x400420` (man beachte: wir betrachten die Adresse des BL-Befehls, nicht den aktuellen Wert von **PC!**, der auch ein anderer sein könnte!). Die Zieladresse ist also `0x400420`.

Für mehr Informationen zu Relocs in ARM64 siehe: [*ELF für die ARM 64-Bit-Architektur (AArch64)*, (2013)][173].

1.31 MIPS-spezifische Details

1.31.1 Laden einer 32-Bit-Konstante in ein Register

```
unsigned int f()
{
        return 0x12345678;
};
```

Alle Anweisungen in MIPS sind genau wie bei ARM 32-Bit groß. Somit ist es nicht möglich eine 32-Bit-Variable in einer Anweisung unterzubringen.

Dementsprechen müssen mindestens zwei Anweisungen genutzt werden: die erste läd den höherwertigen Teil der 32-Bit-Zahl und die zweite führt eine ODER-Anweisung aus, welche den niederwertigen 16-Bit-Teil des Zielregisters setzt:rget register:

Listing 1.389: GCC 4.4.5 -O3 (Assemblercode)

```
        li      $2,305397760  # 0x12340000
        j       $31
        ori     $2,$2,0x5678 ; branch delay slot
```

IDA kennt dieses oft genutzte Muster sehr gut. Aus Gründen der Übersichtlichkeit wird hier die letzte `ORI`-Anweisung als `LI`-Pseudo-Anweisung angezeigt, welche vermeintlich eine komplette 32-Bit-Zahl in das $V0-Register läd.

Listing 1.390: GCC 4.4.5 -O3 (IDA)

```
        lui     $v0, 0x1234
        jr      $ra
        li      $v0, 0x12345678 ; branch delay slot
```

Die GCC-Assembler-Ausgabe beinhaltet ebenfalls die LI-Pseudo-Anweisung, allerdings ist hier auch die Anweisung LUI („Load Upper Immediate“), die einen 16-Bit-Wert im höherwertigen Teil des Registers sichert.

Nachfolgend die *objdump*-Ausgabe:

Listing 1.391: objdump

```
00000000 <f>:
   0:   3c021234        lui     v0,0x1234
```

[173]Auch verfügbar als `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0056b/IHI0056B_aaelf64.pdf`

```
   4:   03e00008        jr      ra
   8:   34425678        ori     v0,v0,0x5678
```

Laden einer globalen 32-Bit-Variable in ein Register

```
unsigned int global_var=0x12345678;

unsigned int f2()
{
        return global_var;
};
```

Dies ist leicht unterschiedlich: LUI läd die oberen 16 Bit von *global_var* in $2 (or $V0). Anschließend läd LW die unteren 16 Bit und addiert sie mit dem Inhalt von $2:

Listing 1.392: GCC 4.4.5 -O3 (Assemblercode)

```
f2:
        lui     $2,%hi(global_var)
        lw      $2,%lo(global_var)($2)
        j       $31
        nop     ; branch delay slot

        ...

global_var:
        .word   305419896
```

IDA kennt auch das oft genutzte Anweisungspaar LUI/LW und fasst beide in der einzelten Anweisung LW zusammen:

Listing 1.393: GCC 4.4.5 -O3 (IDA)

```
_f2:
                lw      $v0, global_var
                jr      $ra
                or      $at, $zero      ; branch delay slot

                ...

                .data
                .globl global_var
global_var:     .word 0x12345678         # DATA XREF: _f2
```

Die Ausgabe von *objdump* ist die selbe wie die Assembler-Ausgabe von GCC. Nachfolgend die Speicherauszüge der Objektdateien:

Listing 1.394: objdump

```
objdump -D filename.o

...
```

```
0000000c <f2>:
   c:   3c020000        lui     v0,0x0
  10:   8c420000        lw      v0,0(v0)
  14:   03e00008        jr      ra
  18:   00200825        move    at,at   ; branch delay slot
  1c:   00200825        move    at,at

Disassembly of section .data:

00000000 <global_var>:
   0:   12345678        beq     s1,s4,159e4 <f2+0x159d8>

...

objdump -r filename.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000c R_MIPS_HI16       global_var
00000010 R_MIPS_LO16       global_var

...
```

Es ist zu sehen, dass die Adresse von *global_var* direkt mit den Anweisungen LUI und LW beim Laden der ausführbaren Datei geschrieben wird: der höherwertige 16-Bit-Teil von *global_var* mit der ersten Anweisung (LUI), der niederwertige 16-Bit-Teil mit der zweiten Anweisung (LW).

1.31.2 Weitere Literatur über MIPS

Dominic Sweetman, *See MIPS Run, Second Edition*, (2010).

Kapitel 2

Wichtige Grundlagen

Kapitel 3

Fortgeschrittenere Beispiele

3.1 strstr()-Beispiel

Erinnern wir uns an die Tatsache, dass GCC manchmal Teile einer Zeichenkette nutzen kann: **??** on page ??.

Die *strstr()* C/C++-Standard-Bibliotheksfunktion wird genutzt um das Auftreten einer Zeichenkette in einer anderen zu finden. Nachfolgend ein Beispiel für die Anwendung:

```
#include <string.h>
#include <stdio.h>

int main()
{
        char *s="Hello, world!";
        char *w=strstr(s, "world");

        printf ("%p, [%s]\n", s, s);
        printf ("%p, [%s]\n", w, w);
};
```

Die Ausgabe ist:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```

Der Unterschied zwischen der Adresse der Original-Zeichenkette und der Adresse des Substrings die *strstr()* zurück gibt ist 7. Tatsächlich hat „Hello, “ ja auch eine Länge von sieben Zeichen.

Der zweite `printf()`-Aufruf weiß nicht, dass weitere Zeichen vor der Zeichenkette sind und gibt lediglich die Zeichen von der Mitte der Original-Zeichenkette bis zum Ende aus (markiert durch ein Null-Byte).

Kapitel 4

Java

4.1 Java

4.1.1 Einführung

Es gibt einige bekannte Decompiler für Java (oder JVM-Bytecode allgemein)[1].

Der Grund ist, dass das Dekompilieren von JVM-Bytecode einfacher ist als von Low-Level x86-Code:

- Es gibt sehr viel mehr Informationen über die Datentypen.
- Das JVM-Speichermodell ist sehr viel strenger und genauer beschrieben.
- Der Java-Compiler führt keine Optimierungen durch (dies macht der JVM JIT[2] während der Laufzeit, so dass der Bytecode in der Klassendatei normalerweise gut lesbar ist).

Wann kann das Wissen über JVM nützlich sein?

- Quick-and-dirty-Patchen von Klassendateien ohne das Neukompilieren der Decompiler-Ergebnisse.
- Analysieren von obfuskiertem Code.
- Erstellen eines eigenen Obfuscators.
- Erstellen eines Compiler Code-Generators (back-end) mit dem Ziel JVM (wie Scala, Clojure, usw.) [3]).

Starten wir mit einigen einfachen Code-Beispielen. Wenn nicht anders erwähnt wird JDK 1.7 verwendet.

Das folgende Kommando wird verwendet um Klassen zu decompilieren:

`javap -c -verbose.`

[1] Beispielsweise JAD: http://varaneckas.com/jad/
[2] Just-In-Time compilation
[3] vollständige Liste: http://en.wikipedia.org/wiki/List_of_JVM_languages

Das folgende Buch habe ich während der Vorbereitung der Beispiele genutzt [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [4].

4.1.2 Rückgabe eines Wertes

Die vermutlich einfachste Java-Funktion ist eine, die einen Wert zurück gibt.

Es sollte hier noch beachtet werden, dass es keine „freien" Funktionen im allgemeinen Sinne in Java gibt sondern „Methoden".

Jede Methode gehört zu einer Klasse, somit ist es nicht möglich eine Methode außerhalb einer Klasse zu definieren.

Wir werden die Methoden hier trotzdem der Einfachheit halber „Funktionen" nennen.

```
public class ret
{
        public static int main(String[] args)
        {
                return 0;
        }
}
```

Kompilieren wir diesen Code:

```
javac ret.java
```

...und dekompilieren ihn mit dem Standard-Java-Tool:

```
javap -c -verbose ret.class
```

Und wir bekommen:

Listing 4.1: JDK 1.7 (excerpt)

```
  public static int main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iconst_0
         1: ireturn
```

Die Java-Entwickler entschieden, dass 0 eine der beliebteste Konstanten in der Programmierung ist, also gibt es eine separate, kurze Ein-Byte-Anweisung die 0 pushed.

4.1.3 Einfache Berechnungsfunktionen

4.1.4 JVM-Speichermodell

x86 und andere low-level Umgebungen nutzen den Stack um Funktionsargumente zu übergeben und lokale Variablen zu speichern.

[4] Auch verfügbar als https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/

JVM behandelt dies ein wenig anders.

- Local variable array (LVA[5]). Dieses wird als Speicher für Funktionsargumente und lokale Variablen genutzt.

  Anweisungen wie `iload_0` laden Werte vom LVA.

  `istore` legt Werte in den Speicher des LVA. Als erstes werden die Funktionsargumente gespeichert, welche mit dem Index 0 oder 1 (Falls das nullte Argument der *this* Pointer ist) beginnen.

  Danach werden die lokalen Variablen alloziert.

  Jeder Eintrag besitzt eine Größe von 32-bit.

  Dadurch benötigen bestimmte Datentypen wie *long* und *double* zwei Einträge.

- Operand stack (or just „stack"). Dieser wird für Berechnungen und die Übergabe von Argumenten genutzt, während andere Funktionen aufgerufen werden.

  Anders als in low-level Umgebungen wie x86, ist es nicht möglich Zugriffe auf den Stack zu machen ohne dabei Anweisungen zu nutzen, welche explizit Werte vom Stack nehmen oder auf den Stack legen.

- Heap. Dieser wird als Speicher für Objekte und Arrays genutzt.

Diese 3 genannten Bereiche sind voneinander isoliert.

4.1.5 Einfache Funktionsaufrufe

4.1.6 Aufrufen von beep()

Dies ist ein einfacher Aufruf zweier Funktionen ohne Argumente:

```
        public static void main(String[] args)
        {
                java.awt.Toolkit.getDefaultToolkit().beep();
        };
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: invokestatic  #2      // Method java/awt/Toolkit.⤦
    ↳ getDefaultToolkit:()Ljava/awt/Toolkit;
         3: invokevirtual #3      // Method java/awt/Toolkit.beep:()V
         6: return
```

Zuerst ruft `invokestatic` bei Offset 0 `java.awt.Toolkit.getDefaultToolkit()` auf, was eine Referenz auf ein Objekt der Klasse `Toolkit` zurück gibt. Die `invokevirtual`-Anweisung bei Offset 3 ruft die `beep()`-Methode dieser Klasse auf.

[5] (Java) Local Variable Array

4.1.7 Linearer Kongruenzgenerator PRNG

4.1.8 Bedingte Sprünge

4.1.9 Argumente übergeben

4.1.10 Bit-Felder

4.1.11 Schleifen

4.1.12 switch()

4.1.13 Arrays

4.1.14 Zeichenketten

4.1.15 Klassen

Einfache Klassen:

Listing 4.2: test.java

```
public class test
{
        public static int a;
        private static int b;

        public test()
        {
            a=0;
            b=0;
        }
        public static void set_a (int input)
        {
                a=input;
        }
        public static int get_a ()
        {
                return a;
        }
        public static void set_b (int input)
        {
                b=input;
        }
        public static int get_b ()
        {
                return b;
        }
}
```

Der Konstruktor setzt lediglich beide Felder auf 0:

```
  public test();
    flags: ACC_PUBLIC
```

```
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1         // Method java/lang/Object."<init>":()⤦
    ⤷ V
         4: iconst_0
         5: putstatic     #2         // Feld a:I
         8: iconst_0
         9: putstatic     #3         // Feld b:I
        12: return
```

Setter von a:

```
  public static void set_a(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic     #2         // Feld a:I
         4: return
```

Getter von a:

```
  public static int get_a();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic     #2         // Feld a:I
         3: ireturn
```

Setter von b:

```
  public static void set_b(int);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=1, args_size=1
         0: iload_0
         1: putstatic     #3         // Feld b:I
         4: return
```

Getter von b:

```
  public static int get_b();
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: getstatic     #3         // Feld b:I
         3: ireturn
```

Es gibt keinen Unterschied in den Codes die mit public oder private Feldern arbeiten.

Aber diese Information ist in der `.class`-Datei vorhanden und es ist nicht möglich auf die privaten Felder zuzugreifen.

Erstellen wir ein Objekt und rufen seine Methoden auf:

Listing 4.3: ex1.java

```
public class ex1
{
        public static void main(String[] args)
        {
                test obj=new test();
                obj.set_a (1234);
                System.out.println(obj.a);
        }
}
```

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
         0: new           #2        // Klasse test
         3: dup
         4: invokespecial #3        // Methode test."<init>":()V
         7: astore_1
         8: aload_1
         9: pop
        10: sipush        1234
        13: invokestatic  #4        // Methode test.set_a:(I)V
        16: getstatic     #5        // Feld java/lang/System.out:Ljava/io/↙
    ↳ PrintStream;
        19: aload_1
        20: pop
        21: getstatic     #6        // Feld test.a:I
        24: invokevirtual #7        // Methode java/io/PrintStream.println↙
    ↳ :(I)V
        27: return
```

Die new-Anweisung erstellt ein Objekt aber ruft den Konstruktor nicht auf (dieser wird bei Offset 4 aufgerufen).

Die `set_a()`-Methode wird an Offset 16 aufgerufen.

Auf das Feld a wird mit der `getstatic`-Anweisung an Offset 21 zugegriffen.

4.1.16 Einfaches Patchen

4.1.17 Zusammenfassung

Was fehlt in Java im Vergleich zu C/C++?

- Strukturen: Nutzen Sie Klassen.
- Unions: Nutzen Sie Klassenhierarchien.
- Vorzeichenlose Datentypen. Dies macht es etwas schwieriger kryptografische Algorithmen in Java zu implementieren.

- Funktionszeiger.

Kapitel 5

Finden von wichtigen / interessanten Stellen im Code

Minimalismus ist kein beliebtes Feature moderner Software.

Aber nicht weil die Programmierer so viel Code schreiben, sondern weil die Libaries allgemein statisch zu ausführbaren Dateien gelinkt werden. Wenn alle externen Libraries in externe DLL Dateien verschoben werden würden, wäre die Welt ein anderer Ort. (Ein weiterer Grund für C++ sind die STL[1] und andere Template-Libraries.)

Deshalb ist es sehr wichtig den Ursprung einer Funktion zu bestimmen, wenn die Funktion aus einer Standard-Library oder aus einer sehr bekannten Library stammt (wie z.B Boost[2], libpng[3]), oder ob die Funktion sich auf das bezieht was wir im Code versuchen zu finden.

Es ist ein wenig absurd sämtlichen Code in C/C++ neu zu schreiben, um das zu finden was wir suchen.

Eine der Hauptaufgaben eines Reverse Enigneers ist es schnell Code zu finden den er/sie sucht.

Der IDA-Disassembler erlaubt es durch Textstrings, Byte-Sequenzen und Konstanten zu suchen. Es ist sogar möglich den Code in .lst oder .asm Text Dateien zu exportieren und diese mit `grep`, awk, etc. zu untersuchen.

Wenn man versucht zu verstehen wie ein bestimmter Code funktioniert, kann auch eine einfache Open-Source-Library wie libpng als Beispiel dienen. Wenn man also eine Konstante oder Textstrings findet die vertraut erscheinen, ist es immer einen Versuch wert diese zu *googlen* . Und wenn man ein Opensource Projekt findet in dem diese Funktion benutzt wird, reicht es meist aus diese Funktionen miteinander zu vergleichen. Es könnte helfen Teile des Problems zu lösen.

[1](C++) Standard Template Library

[2]`http://www.boost.org/`

[3]`http://www.libpng.org/pub/png/libpng.html`

Zum Beispiel, wenn ein Programm XML Dateien benutzt, wäre der erste Schritt zu ermitteln welche XML-Library benutzt wird für die Verarbeitung, da die Standard (oder am weitesten verbreitete) libraries normal benutzt werden anstatt selbst geschriebene librarys.

Zum Beispiel, der Autor dieser Zeilen wollte verstehen wie die Kompression/Dekompression von Netzwerkpaketen in SAP 6.0 funktioniert. SAP ist ein gewaltiges Stück Software, aber detaillierte -PDB Dateien mit Debug Informationen sind vorhanden, was sehr praktisch ist. Der Autor hat schließlich eine Ahnung gehabt, das eine Funktion genannt *CsDecomprLZC* die Dekompression der Netzwerkpakete übernahm. Er hat nach dem Namen der Funktion auf google gesucht und ist schnell zum schluss gekommen das diese Funktion in MaxDB benutzt wurde (Das ist ein Open-Source SAP Projekt) [4].

`http://www.google.com/search?q=CsDecomprLZC`

Erstaunlich, das MaxDB und die SAP 6.0 Software den selben Code geteilt haben für die Kompression/Dekompression der Netzwerkpakete.

5.1 Ausführbare Dateien Identifizieren

5.1.1 Microsoft Visual C++

MSVC Versionen und DLLs die Importiert werden können:

Marketing ver.	Internal ver.	CL.EXE ver.	DLLs imported	Release date
6	6.0	12.00	msvcrt.dll msvcp60.dll	June 1998
.NET (2002)	7.0	13.00	msvcr70.dll msvcp70.dll	February 13, 2002
.NET 2003	7.1	13.10	msvcr71.dll msvcp71.dll	April 24, 2003
2005	8.0	14.00	msvcr80.dll msvcp80.dll	November 7, 2005
2008	9.0	15.00	msvcr90.dll msvcp90.dll	November 19, 2007
2010	10.0	16.00	msvcr100.dll msvcp100.dll	April 12, 2010
2012	11.0	17.00	msvcr110.dll msvcp110.dll	September 12, 2012
2013	12.0	18.00	msvcr120.dll msvcp120.dll	October 17, 2013

msvcp*.dll hat C++-bezogene Funktionen, bedeutet wenn die library importiert wird, ist das Programm das sie importiert wahrscheinlich ein C++ program.

[4] Mehr darüber in der relevanten Sektion (**??** on page ??)

Name mangling

Die Namen fangen normal an mit dem ? Symbol.

Hier: **??** on page ?? kann man mehr lesen über MSVC's Name Mangling .

5.1.2 GCC

Neben *NIX Umgebungen, ist GCC auch in win32 Umgebungen präsent, in der Form von Cygwin and MinGW.

Name mangling

Namen fangen hier normal mit dem _Z Symbolen an.

Man kann mehr lesen über GCC's Name Mangling hier: **??** on page ??.

Cygwin

cygwin1.dll wird oft importiert.

MinGW

msvcrt.dll wird vielleicht importiert.

5.1.3 Intel Fortran

libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP Support) werden vielleicht importiert.

libifcoremd.dll hat eine menge an Funktionen die das `for_` Präfix haben, was *Fortran* bedeutet.

5.1.4 Watcom, OpenWatcom

Name mangling

Namen fangen normal mit dem `W` Symbol an.

Zum Beispiel wird so eine Methode benannt „method" der Klasse „class" die keine Argumente hat und *void* zurück gibt:

```
W?method$_class$n__v
```

5.1.5 Borland

Hier ist ein Beispiel für Borland Delphi's und C++Builder's Name Mangling:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

Die Namen fangen immer mit dem @ Symbol an, dann haben wir den Namen der Klassen Namen, Methoden Namen, und codiert die Typen der Argumente der Methode.

Diese Namen können in den .exe Imports, .dll Exports, Debug Daten und etc existieren.

Borland Visual Component Libraries (VCL) werden in .bpl Dateien gehalten anstatt .dll's, zum Beispiel vcl50.dll, rtl60.dll.

Eine weitere DLL die vielleicht importiert wird: BORLNDMM.DLL

Delphi

Fast alle Delphi executables haben den „Boolean" Text String am Anfang des Code Segments, zusammen mit den Namen anderer Typen liegen.

Dies ist ein sehr typischer Anfang für das CODE Segment bei einem Delphi Programm, dieser Block kam direkt nach dem win32 PE Datei header:

```
04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00  |..@...Boolean...|
00 00 01 00 00 00 00 10 40 00 05 46 61 6c 73 65  |........@..False|
04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69  |.True.@.,.@...Wi|
64 65 43 68 61 72 03 00 00 00 00 ff ff 00 00 90  |deChar..........|
44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff  |D.@...Char......|
00 00 00 90 58 10 40 00 01 08 53 6d 61 6c 6c 69  |....X.@...Smalli|
6e 74 02 00 80 ff ff ff 7f 00 00 90 70 10 40 00  |nt..........p.@.|
01 07 49 6e 74 65 67 65 72 04 00 00 00 80 ff ff  |..Integer.......|
ff 7f 8b c0 88 10 40 00 01 04 42 79 74 65 01 00  |......@...Byte..|
00 00 00 ff 00 00 00 90 9c 10 40 00 01 04 57 6f  |..........@...Wo|
72 64 03 00 00 00 00 ff ff 00 00 90 b0 10 40 00  |rd............@.|
01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 00 ff  |..Cardinal......|
ff ff ff 90 c8 10 40 00 10 05 49 6e 74 36 34 00  |......@...Int64.|
00 00 00 00 00 00 80 ff ff ff ff ff ff ff 7f 90  |................|
e4 10 40 00 04 08 45 78 74 65 6e 64 65 64 02 90  |..@...Extended..|
f4 10 40 00 04 06 44 6f 75 62 6c 65 01 8d 40 00  |..@...Double..@.|
04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90  |..@...Currency..|
14 11 40 00 0a 06 73 74 72 69 6e 67 20 11 40 00  |..@...string .@.|
0b 0a 57 69 64 65 53 74 72 69 6e 67 30 11 40 00  |..WideString0.@.|
0c 07 56 61 72 69 61 6e 74 8d 40 00 40 11 40 00  |..Variant.@.@.@.|
0c 0a 4f 6c 65 56 61 72 69 61 6e 74 98 11 40 00  |..OleVariant..@.|
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 98 11 40 00  |..............@.|
04 00 00 00 00 00 00 00 18 4d 40 00 24 4d 40 00  |.........M@.$M@.|
28 4d 40 00 2c 4d 40 00 20 4d 40 00 68 4a 40 00  |(M@.,M@. M@.hJ@.|
84 4a 40 00 c0 4a 40 00 07 54 4f 62 6a 65 63 74  |.J@..J@..TObject|
a4 11 40 00 07 07 54 4f 62 6a 65 63 74 98 11 40  |..@...TObject..@|
00 00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00  |........System..|
c4 11 40 00 0f 0a 49 49 6e 74 65 72 66 61 63 65  |..@...IInterface|
00 00 00 00 01 00 00 00 00 00 00 00 c0 00 00  |................|
00 00 00 00 46 06 53 79 73 74 65 6d 03 00 ff ff  |....F.System....|
f4 11 40 00 0f 09 49 44 69 73 70 61 74 63 68 c0  |..@...IDispatch.|
11 40 00 01 00 04 02 00 00 00 00 00 c0 00 00 00  |.@..............|
00 00 00 46 06 53 79 73 74 65 6d 04 00 ff ff 90  |...F.System.....|
cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8  |..D$...Ql...D$..|
e9 6f 6c 00 00 83 44 24 04 f8 e9 79 6c 00 00 cc  |.ol...D$...yl...|
cc 21 12 40 00 2b 12 40 00 35 12 40 00 01 00 00  |.!.@.+.@.5.@....|
00 00 00 00 00 00 00 00 00 c0 00 00 00 00 00 00  |................|
46 41 12 40 00 08 00 00 00 00 00 00 00 8d 40 00  |FA.@..........@.|
bc 12 40 00 4d 12 40 00 00 00 00 00 00 00 00 00  |..@.M.@.........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
bc 12 40 00 0c 00 00 00 4c 11 40 00 18 4d 40 00  |..@.....L.@..M@.|
50 7e 40 00 5c 7e 40 00 2c 4d 40 00 20 4d 40 00  |P~@.\~@.,M@. M@.|
6c 7e 40 00 84 4a 40 00 c0 4a 40 00 11 54 49 6e  |l~@..J@..J@..TIn|
74 65 72 66 61 63 65 64 4f 62 6a 65 63 74 8b c0  |terfacedObject..|
d4 12 40 00 07 11 54 49 6e 74 65 72 66 61 63 65  |..@...TInterface|
64 4f 62 6a 65 63 74 bc 12 40 00 a0 11 40 00 00  |dObject..@...@..|
00 06 53 79 73 74 65 6d 00 00 8b c0 00 13 40 00  |..System......@.|
11 0b 54 42 6f 75 6e 64 41 72 72 61 79 04 00 00  |..TBoundArray...|
00 00 00 00 00 03 00 00 00 6c 10 40 00 06 53 79  |.........l.@..Sy|
73 74 65 6d 28 13 40 00 04 09 54 44 61 74 65 54  |stem(.@...TDateT|
69 6d 65 01 ff 25 48 e0 c4 00 8b c0 ff 25 44 e0  |ime..%H......%D.|
```

Die ersten 4 Btyes des Daten Segments (DATA) können `00 00 00 00`, `32 13 8B C0` oder FF FF FF FF sein.

Diese Informationen können nützlich sein wenn man mit gepackten oder verschlüsselten Delphi executables arbeiten muss.

5.1.6 Other known DLLs

- vcomp*.dll—Microsoft's Implementierung von OpenMP.

5.2 Kommunikation mit der außen Welt (Funktion Level)

Oft ist es empfehlenswert die Funktionsargumente und die Rückgabewerte im Debugger oder DBI[5] zu überwachen. Zum Beispiel hat der Autor einmal versucht die

[5] Dynamic Binary Instrumentation

Bedeutung einer obskuren Funktion zu verstehen, die einen inkorrekten Bubblesort-Algorithmus implementiert hatte[6] (Sie hat funktioniert, jedoch viel langsamer als normal). Die Eingaben und Ausgaben zur Laufzeit der Funktion zu überwachen hilft sofort zu verstehen was die Funktion tut.

5.3 Kommunikation mit der Außen Welt (Win32)

Manchmal reicht es die Ein- und Ausgaben einer Funktion zu beobachten um zu verstehen was sie tut. Auf diese Weise kann man Zeit Sparren.

Datei und Regestry Zugriff: Für einfache Analysen kann das Tool Prozess Monitor[7] von SysInternals hilfreich sein.

Bei einfachen Netzwerk Zugriffen ist Wireshark[8] zum analysieren ganz nützlich.

Trotzdem muss man einen Blick in die Netzwerkpakte werfen.
Das erste wonach man schauen kann ist welche Funktionen die BS API[9]s benutzen und was für Standard libraries benutzt werden.

Wenn das Programm unterteilt ist in eine Main executable und mehrere DLL Dateien, können manchmal die Namen der Funktionen innerhalb der DLLs Helfen.

Wenn wir daran interessiert sind was genau zum Aufruf von `MessageBox()` mit einem spezifischen Text führt, können wir versuchen diesen Text innerhalb des Data Segments zu finden, die Referenzen auf den Text und die Punkte von denen aus die Kontrolle an den `MessageBox()` Aufruf an dem wir interessiert sind.

Wenn wir über Video Spiele sprechen sind wir daran interessiert welche `rand()` aufrufe mehr oder weniger zufällig darin vorkommen, vielleicht versuchen wir die `rand()` Funktion oder Ersatz Funktionen zu finden (wie z.B der Mersenne Twister Algorithmus) und wir versuchen die Orte zu finden von welchen aus diese Funktionen aufgerufen werden, und noch wichtiger was für Ergebnisse verwertet werden. Ein Beispiel: **??** on page ??.

Aber wenn es sich nicht um ein Spiel handelt und `rand()` wird trotzdem benutzt, ist es interessant zu wissen warum. Es gibt Fälle bei denen unerwartet `rand()` in Daten Kompressions Algorithmen benutzt wird (für die Imitation von Verschlüsslung): blog.yurichev.com.

5.3.1 Oft benutzte Funktionen in der Windows API

Diese Funktionen sind vielleicht unter den importierten. Es ist Sinnvoll an dieser Stelle zu erwähnen das nicht unbedingt jede Funktion benutzt wird aus dem Code den der Programmierer geschrieben hat.

Manche Funktionen haben eventuell das `-A` Suffix für die ASCII Version und das `-W` für die Unicode Version.

[6] https://yurichev.com/blog/weird_sort_KLEE/
[7] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
[8] http://www.wireshark.org/
[9] Application Programming Interface

- Registry zugriff (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx.
- Zugriff auf text .ini-files (kernel32.dll): GetPrivateProfileString.
- Dialog boxes (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText.
- Resourcen zugriff (6.5.2 on page 620): (user32.dll): LoadMenu.
- TCP/IP networking (ws2_32.dll): WSARecv, WSASend.
- Datei Zugriff (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx.
- High-level Zugriff auf das Internet (wininet.dll): WinHttpOpen.
- Die digitale Signatur einer ausführbaren Datei prüfen (wintrust.dll):

  WinVerifyTrust.
- Die Standard MSVC library (wenn sie dynamisch gelinked wurde)

  assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

5.3.2 Verlängerung der Testphase

Registry Zugriffs Funktionen sind häufig ziele für Leute die versuchen die Testphase einer Software zu cracken, die eventuell die Installations Zeit und Datum in der Regestry zu speichert.

Ein weiteres beliebtes Ziel sind die GetLocalTime() und GetSystemTime() Funktionen: eine Test Software, muss bei jedem Start die aktuelle Zeit und Datum überprüfen.

5.3.3 Entfernen nerviger Dialog Boxen

Ein verbreiteter Weg raus zu finden was eine dieser nervigen Dialog boxen macht, ist den Aufruf von MessageBox(), CreateDialog() und CreateWindow() Funktionen abzufangen.

5.3.4 tracer: Alle Funktionen innerhalb eines bestimmten Modules abfangen

Es gibt INT3 breakpoints in tracer, die nur einmal ausgelöst werden. Jedoch können diese breakpoints für alle Funktionen in einer bestimmten DLL gesetzt werden.

```
--one-time-INT3-bp:somedll.dll!.*
```

Oder, lasst uns einfach mal INT3 breakpoints für alle Funktionen setzen, die das `xml` Präfix in ihrem Namen haben:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

Die andere Seite der Medaille ist, solche breakpoints werden nur einmal ausgelöst. Tracer zeigt den Aufruf der Funktion, wenn er passiert, aber auch nur einmal. Ein weiterer Nachteil ist—es ist unmöglich die Argumente der Funktion zu betrachten.

Dennoch, dieses Feature ist sehr nützlich wenn man weiß das das Programm eine DLL benutzt, aber man nicht weiß welche Funktionen aufgerufen werden. Und es gibt eine ganze Menge an Funktionen.

Zum Beispiel, schauen wir uns einmal an was das uptime Kommando aus cygwin benutzt:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Dadurch sehen wir alle cygwin1.dll library Funktionen die zumindest einmal aufgerufen wurden, und von welcher Stelle:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0⤦
    ↳ x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!⤦
    ↳ OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP⤦
    ↳ +0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!⤦
    ↳ OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP⤦
    ↳ +0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0⤦
    ↳ x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP⤦
    ↳ +0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0⤦
    ↳ x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP⤦
    ↳ +0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0⤦
    ↳ x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0⤦
    ↳ x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0⤦
    ↳ x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0⤦
    ↳ x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP⤦
    ↳ +0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0⤦
    ↳ x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP⤦
    ↳ +0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP⤦
    ↳ +0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP⤦
    ↳ +0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0⤦
    ↳ x4c3 (0x4014c3))
```

5.4 Strings

5.4.1 Text strings

C/C++

Die normalen C-strings sind NULL-Terminiert (ASCIIZ-strings).

Der Grund warum C Stringformatierung so ist wie sie ist (NULL-Terminiert) scheint ein Historischer zu sein. In [Dennis M. Ritchie, *The Evolution of the Unix Time-sharing System*, (1979)] kann man nach lesen:

> Ein kleiner Unterschied war das die I/O Einheit ein "word" war, nicht ein Byte, weil die PDP-7 eine word-adressierte Maschine war. In der Praxis bedeutete das lediglich das alle Programme die mit Zeichen Streams arbeiteten, das NULL Zeichen ignorieren mussten, weil die NULL benutzt wurde um eine Datei bis zu einer Graden Zahl an Bytes auf zu füllen.

In Hiew oder FAR Manager sehen diese Strings so aus:

```
int main()
{
        printf ("Hello, world!\n");
};
```

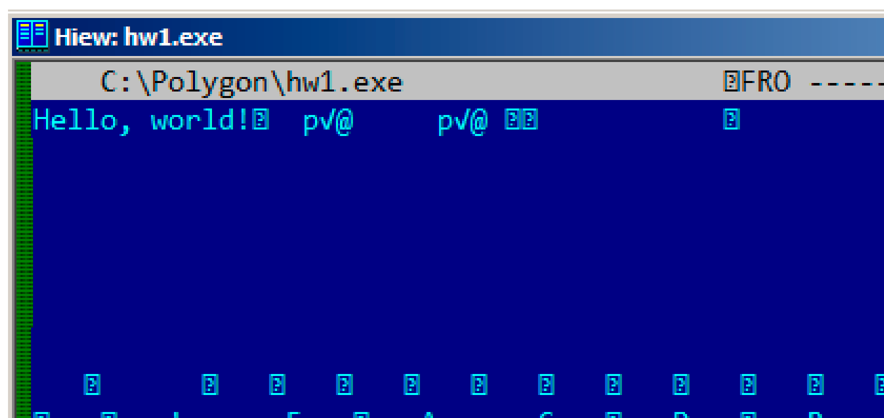


Abbildung 5.1: Hiew

Borland Delphi

Dem String in Passcal und Borland Delphi hängt eine 8 oder 32-Bit Zeichenkette an.

Zum Beispiel:

Listing 5.1: Delphi

```
CODE:00518AC8                  dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0

...

CODE:00518AFC                  dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

Unicode

Oft, ist das was Unicode genannt wird einfach eine Methode um Strings zu codieren, bei denen jedes Zeichen 2 Byte oder 16 Bits verbraucht. Das ist ein hüfiger Terminologischer Fehler. Unicode ist ein Standard bei dem eine Nummer zu einem der vielen Schreibsysteme der Welt zugeordnet wird, aber es beschreibt nicht die codierungs Methode.

Die bekannteste Methode zu Codieren ist: UTF-8 (ist weit verteilt im Internet und auf *NIX Systemen) und UTF-16LE (wird bei Windows benutzt).

UTF-8

UTF-8 ist eine der erfolgreichsten Methoden um Zeichen zu codieren. Alle Latein Zeichen werden codiert so wie in ASCII, und alle Symbole nach der ASCII Tabelle wurden codiert mit zusätzlichen Bytes. 0 wird codiert als davor, also arbeiten alle Standard C String Funktionen mit UTF-8 Strings wie mit jedem anderen String auch.

Lasst uns anschauen wie die Symbole in verschiedenen anderen Sprachen nach UTF-8 Codiert werden und wie man sie als FAR aussehen lassen kann, durch das benutzen der codepage 437[10]:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

[10]Beispiel und Übersetzung kø”nnen von hier bezogen werden: http://www.columbia.edu/~fdc/utf8/

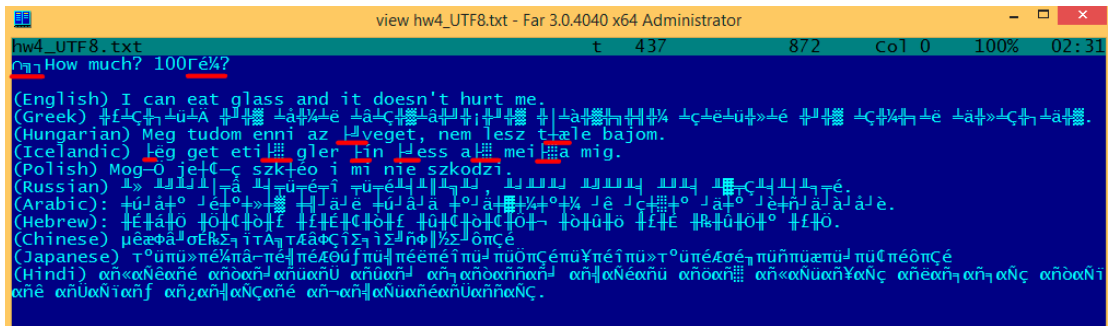


Abbildung 5.2: FAR: UTF-8

Wie man hier sehen kann, der Englische String sieht genauso aus wie sein Gegenstück in ASCII.

Die Ungarische Sprache benutzt Latein Symbole plus ein paar Symbole mit diacritic Markierungen.

Diese Symbole werden mit mehreren Bytes codiert, diese wurden rot unterstrichen. Das gleiche gilt für die Isländischen und Polnischen Sprachen.

Es gibt auch das „Euro“ Währungs Symbol im Standard, das Symbol wurde mit 3 Bytes Codiert.

Der Rest der Schreibsysteme hat keinen Bezug zu Latein.

Zumindest in Russisch, Arabisch, Hebräisch und Hindu können wir wiederkehrende Bytes erkennen und das ist nicht mal überraschend: Alle Zeichen eines Schreibsystems werden normalerweise in der selben Unicode Tabelle angelegt, also fängt ihr code mit den immer gleichen nummern an.

Zu Anfang, noch vor dem „How much?“ String sehen wir 3 Bytes, die tatsächlich das BOM[11] darstellen. Das BOM definiert das Codierungssystem das benutzt werden soll.

UTF-16LE

Viele win32 Funktionen in Windows haben die Suffixe `-A` und `-W`. Der erste Typ Funktionen arbeitet mit normalen Strings, der andere Typ mit UTF-16LE Strings (*wide*).

Im zweiten Fall, wird jedes Symbol normal als 16-Bit Wert des Typs *short* gespeichert.

Die Latein Symbole in UFT-16 Strings sehen in Hiew oder FAR aus als wären sie mit Null Bytes verschachtelt:

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

[11]Byte Order Mark

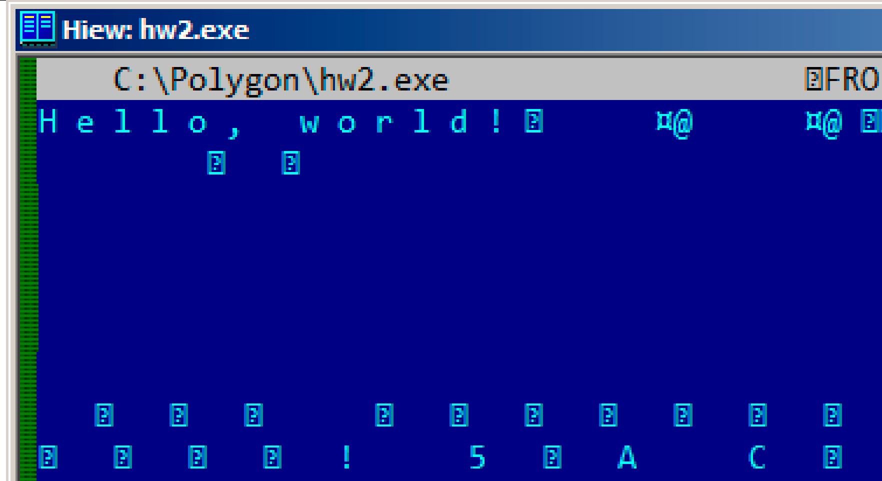


Abbildung 5.3: Hiew

Wir kø”nnen das oft auch in glsWindows NT System Dateien sehen:

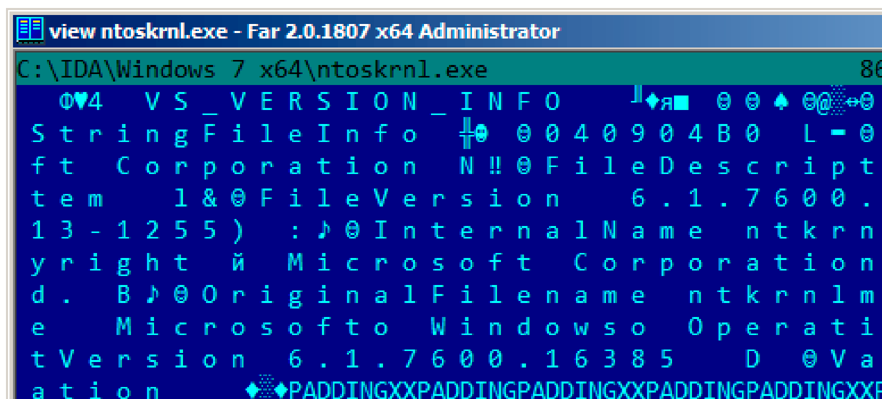


Abbildung 5.4: Hiew

Strings mit Zeichen die exakt 2 Bytes verbrauchen werden „Unicode“ in IDA genannt:

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Hier sieht man wie Russische Sprache in UTF-16LE Codiert wird:

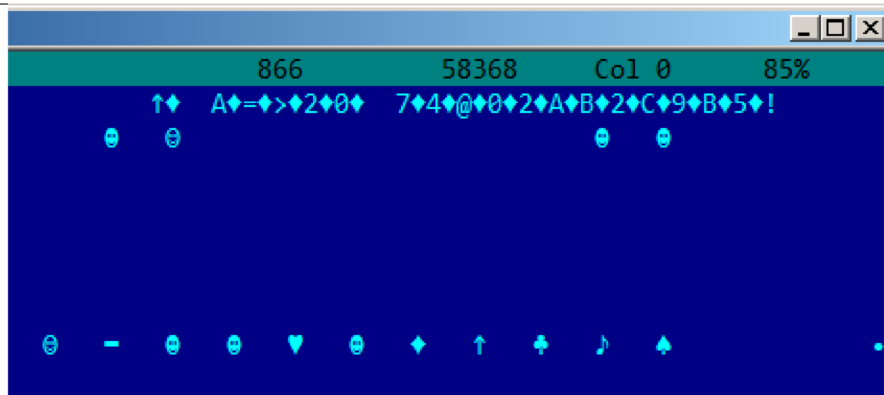


Abbildung 5.5: Hiew: UTF-16LE

Was man leicht sehen kann ist das die Symbole durchzogen sind von den Diamant Zeichen (das im ASCII code mit 4 codiert wird). Tatsächlich, findet man die Kyrillischen Symbole in der vierten Unicode Tabelle. Deswegen, alle Kyrillischen Symbole in UTF-16LE findet man im Bereich `0x400-0x4FF`.

Lass uns noch mal zu dem Beispiel gehen mit dem String der in verschiedenen Sprachen geschrieben ist. Hier sieht man wie der String in UTF-16LE aussieht.

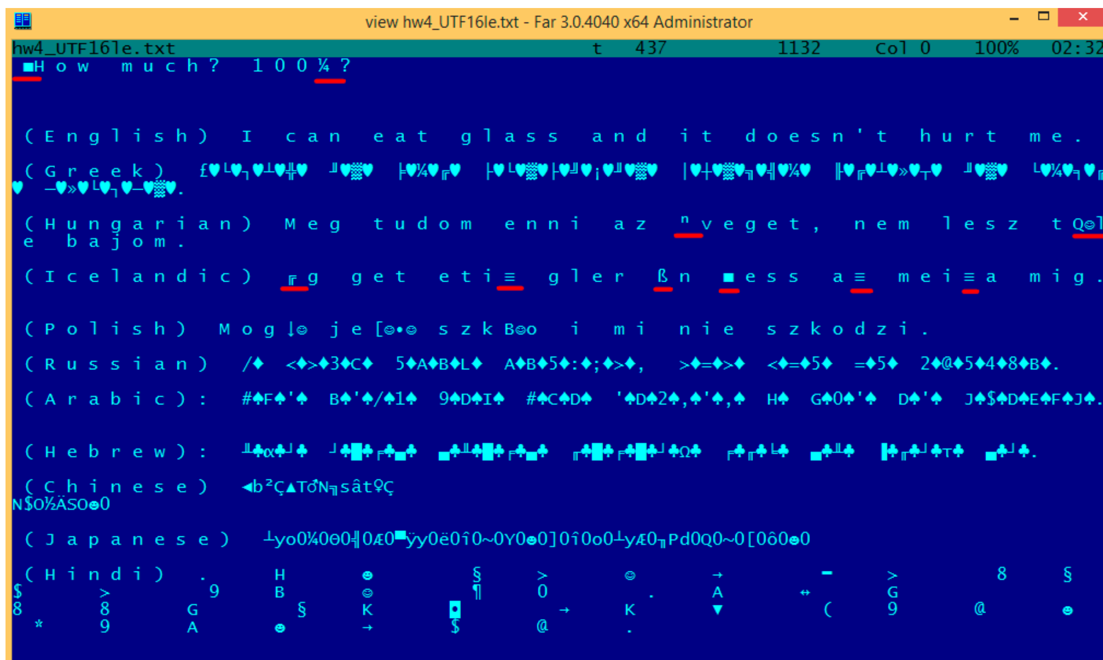


Abbildung 5.6: FAR: UTF-16LE

Hier können wir auch das BOM am Anfang sehen. Alle Latein Zeichen enthalten Null

Bytes.

Manche Zeichen mit unterschiedlichen Markierungen (Ungarisch und Isländisch) wurden rot unterstrichen.

Base64

Die Base64 Codierung ist sehr weit verbreitet für fälle in denen man Binärdaten als Textstring übertragen will.

Im Grunde, codiert dieser Algorithmus 3 Binär Bytes in 4 druckbare Zeichen: Alle 26 Latein Zeichen (beides klein und groß Buchstaben), Ziffern, plus Zeichen („+“) und slash Zeichen („/“), 64 Zeichen insgesamt.

Ein charakteristisches Feature von Base64 Strings ist das sie oft (aber nicht immer) mit 1 oder 2 Padding Gleichheitszeichen („=“) Enden, zum Beispiel:

```
AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=
```

```
WVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuQ==
```

Das Gleichheitszeichen Symbol (q=) wird man niemals in der Mitte eines Base64-codierten Strings sehen.

Jetzt ein Beispiel wie man per Hand Base64 codieren kann. Lasst uns 0x00, 0x11 , 0x22 und 0x33 in Hexadezimalzahlen in einen Base64 String umwandeln:

```
$ echo -n "\x00\x11\x22\x33" | base64
ABEiMw==
```

Lasst uns alle 4 Bytes in Binär Form bringen und dann neu gruppieren in 6-Bit Gruppen:

```
|  00  ||  11  ||  22  ||  33  ||      ||      |
00000000000100010010001000110011????????????????
| A   || B   || E   || i   || M   || w   || =   || =   |
```

Die ersten drei Bytes (0x00, 0x11, 0x22) können in 4 Base64 Zeichen umgewandelt werden (“ABEi”), aber nicht das letzte Byte (0x33), also wird das Byte codiert indem man zwei Buchstaben benutzt (“Mw”) und das Padding Symbol (“=”) wird zweimal hinzugefügt um die letzte Gruppe auf 4 Zeichen zu erweitern. Das bedeutet das die Länge aller korrekten Base64 Strings sich immer durch 4 Teilen lässt.

Base64 wird oft benutzt wenn es darum geht Binärdaten in XML Dateien zu speichern. “Armored” (z.B, in Text Form) PGP Cookie und Signaturen werden codiert mit Base64.

Manche Leute versuchen auch Base64 zu benutzen um Strings zu verschleiern. http://blog.sec-consult.com/2016/01/deliberately-hidden-backdoor-account-in.html [12].

[12] http://archive.is/nDCas

Es gibt Werkzeuge zum scannen von beliebigen Binärdateien nach Base64 Strings. Ein solch ein Scanner ist base64scanner[13].

Ein weiteres Codierungssystem welches im UseNet und FidoNet sehr weit verbreitet war, ist UUencoding. Binärdateien sind in Phrack Magazine immernoch mit UUencoding codiert. Es hat eigentlich die gleichen Features, unterscheidet sich von Base64 jedoch insofern, dass der Dateiname auch im Header gespeichert wird.

By the Way: Es gibt auch einen nahen Verwandten zu Base64: Base32., ein Alphabet das 10 Zeichen und 26 Latein Zeichen hat. Eine verbreitete Anwendung ist Onion Adressen zu codieren. [14], z.B:
http://3g2upl4pq6kufc4m.onion/. URL[15] kann keine mixed-case Latein Zeichen beinhalten, deshalb haben Tor Entwickler sich für Base32 entschieden.

5.4.2 Strings in Binär finden

> Actually, the best form of Unix documentation is frequently running the **strings** command over a program's object code. Using **strings**, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.
>
> The Unix-Haters Handbook

Das Standard UNIX *strings* Utility ist ein quick-n-dirty Weg um alle Strings in der Datei an zu schauen. Zum Beispiel, in der OpenSSH 7.2 sshd executable Datei gibt es einige Strings:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/
%02x
...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fdbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUaHZDlIbQaO2dS9FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
```

[13] https://github.com/DennisYurichev/base64scanner
[14] https://trac.torproject.org/projects/tor/wiki/doc/HiddenServiceNames
[15] Uniform Resource Locator

```
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshrc
...
D$4PQWR1
D$4PUj
D$4PV
D$4PVj
D$4PW
D$4PWj
D$4X
D$4XZj
D$4Y
...
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
digests
D$iPV
direct-streamlocal
direct-streamlocal@openssh.com
...
FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A6...
...
```

Dort kann man Optionen, Fehler Meldungen, Datei Pfade, importierte dynamische Module, Funktionen und einige andere komische Strings (keys?) sehen. Es gibt auch nicht druckbare Zeichen—x86 Code enthält chunks von druckbaren ASCII Zeichen, bis zu ca 8 Zeichen.

Sicher, OpenSSH ist ein open-source Programm. Aber sich die lesbaren Strings eines unbekannten Programms an zuschauen ist meist der erste Schritt bei der Analyse.

grep kann genauso benutzt werden.

Hiew hat die gleichen Fähigkeiten (Alt-F6), genau wie der Sysinternals ProcessMonitor.

5.4.3 Error/debug Narchichten

Debugging Messages sind auch sehr nützlich, wenn vorhanden. Auf gewisse weise, melden die debug Narichten was gerade im Programm vorgeht. Oft schreiben diese `printf()`-ähnlichen Funktionen, in log-Dateien oder sie schreiben nirgends hin aber die calls zu den printf-ähnlichen Funktionen sind noch vorhanden, weil der build kein Debug build aber ein *release* ist.

Wenn lokale oder globale Variablen in Debug messages geschrieben werden, kann das auch hilfreich sein da man so an die Variablen Namen kommt. Zum Beispiel, eine solche Funktion in Oracle RDBMS ist `ksdwrt()`.

Textstrings mit Aussage sind auch Hilfreich. Der IDA disassembler zeigt welche Funktion und von welchem Punkt aus ein spezifischer String benutzt wird. Manchmal passieren lustige Dinge dabei[16].

[16] blog.yurichev.com

Fehlermeldungen helfen uns genauso. In Oracle RDBMS, werden Fehler von einer Gruppe von Funktionen gemeldet. Über das Thema kann man mehr hier erfahren: blog.yurichev.com.

Es ist Möglich heraus zu finden welche Funktionen Fehler melden und unter welchen Bedingungen.

Übrigens, das ist für Kopierschutztsysteme oft der Grund kryptische Fehlermeldungen oder einfach nur Fehlernummer aus zu geben. Niemand ist glücklich darüber wenn der Softwarecracker den Kopierschutz besser versteht nur weil dieser durch eine Fehlermeldung ausgelöst wurde.

Ein Beispiel von verschlüsselten Fehlermeldungen gibt es hier: **??** on page ??.

5.4.4 Verdächtige magic strings

Manche Magic Strings die in Hintertüren benutzt werden sehen schon ziemlich verdächtig aus.

Zum Beispiel, es gab eine Hintertür im TP-Link WR740 Home Router[17]. Die Hintertür konnte aktiviert werden wenn man folgende URL aufrief: `http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html`.

Tatsächlich, kann man den Magic String „userRpmNatDebugRpm26525557“ in der Firmware finden.

Der String war nicht googlebar bis die Information öffentlich über die Hintertür öffentlich verbreitet wurde.

Man würde solche Informationen natürlich auch nicht in irgendeinem RFC[18] finden.

Man würde auch keinen Algorithmus finden der solch seltsame Byte Sequenzen benutzt.

Und es sieht auch nicht nach einer Fehler- order Debugnaricht aus.

Also es ist immer eine gute Idee so seltsamen Dinge genauer zu betrachten.

Manchmal, sind solche Strings auch mit base64 codiert.

Es ist also immer eine gute Idee diese Stings zu Decodieren und sie visuell zu durchsuchen, ein Blick kann schon genügen.

Präziser gesagt, diese Methode Hintertüren zu verstecken nennt man „security through obscurity“.

5.5 assert() Aufrufe

Manchmal ist die Präsenz des `assert()` macro’s ebenfalls nützlich: allgemein erlaubt dieses Makro Rückschlüsse auf source code Dateinamen, Zeilen nummern und die Bedienung für das Makro im Code.

[17] `http://sekurak.pl/tp-link-httptftp-backdoor/`
[18] Request for Comments

Die nützlichste Informationen ist enthalten in der Bedingung von assert, wir können Variablennamen oder Namen von Struct Feldern ableiten. Ein weiteres nützliches Stück Information sind die Datei Namen—Wir können versuchen abzuleiten von welcher Art der Code ist. Es ist ebenfalls möglich bekannte open-source library-Namen von den Datei Namen abzuleiten.

Listing 5.2: Example of informative assert() calls

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ;
    "td->td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

Es ist Empfehlenswert beides die Konditionen und die Datei Namen in „google“ zu suchen, was zu einer open-source library führen kann. Zum Beispiel, wenn wir „sp->lzw_nbits <= BITS_MAX“ in „google“ suchen, ist es absehbar das wir als Ergebnis Code aus der Open-Source library für die LZW Kompression bekommen.

5.6 Konstanten

Menschen, Programmierer eingeschlossen, neigen dazu Zahlen zu runden wie z.B 10, 100, 1000, im realen Leben so wie in ihrem Code.

Der angehende Reverse Engineer kennt diese Werte und ihre hexadezimale Repräsentation sehr gut: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Die Konstanten `0xAAAAAAAA` (0b10101010101010101010101010101010) und

0x55555555 (0b01010101010101010101010101010101) sind auch sehr populär—sie sind zusammengesetzt aus verändernden Bits.

Dies hilft Signale voneinander zu unterscheiden bei denen alle Bits eingeschaltet (0b1111 …) oder ausgeschaltet (0b0000 …) werden . Zum Beispiel wird die Konstante 0x55AA beim Boot Sektor, MBR[19], und im **ROM!** von IBM-Kompatiblen Erweiterung Karten benutzt.

Manche Algorithmen, speziell die Kryptografischen benutzen eindeutige Konstanten, die mit der Hilfe von IDA einfach im Code zu finden sind.

Zum Beispiel, der MD5 Algorithmus initialisiert seine Internen Variablen wie folgt:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Wenn man diese vier Konstanten im Code hintereinander benutzt findet, dann ist die Wahrscheinlichkeit das diese Funktion sich auf MD5 bezieht.

Ein weiteres Beispiel sind die CRC16/CRC32 Algorithmen, ihre Berechnungs Algorithmen benutzen oft vorberechnete Tabellen wie diese:

Listing 5.3: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, 0x0440,
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
        ...
```

Man beachte auch die vorberechnete Tabelle für CRC32: **??** on page ??.

In tabellenlosen CRC-Algorithmen werden bekannte Polynome benutzt, zum Beispiel, 0xEDB88320 für CRC32.

5.6.1 Magic numbers

Viele Datei-Formate definieren einen Standard-Dateiheader in dem eine *magic number(s)* benutzt wird, einzelne oder sogar mehrere.

Zum Beispiel, alle Win32 und MS-DOS executable starten mit zwei Zeichen „MZ“.

Am Anfang einer MIDI Datei muss die „MThd“ Signatur vorhanden sein. Wenn wir ein Programm haben das auf MIDI Dateien zugreift um sonst was zu machen, ist es sehr wahrscheinlich das das Programm die Datei validieren muss in dem es mindestens die ersten 4 Bytes prüft.

Das kann man wie folgt realisieren: (*buf* Zeigt auf den Anfang der geladenen Datei im Speicher)

[19]Master Boot Record

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

…oder durch das Aufrufen der Funktion für das vergleichen von Speicherblöcken wie z.B `memcmp()` oder beliebigen anderen Code bis hin zu einer `CMPSB` (**??** on page ??) Instruktion.

Wenn man so einen Punkt findet kann man bereits sagen das eine MIDI Datei geladen wird, wir können auch sehen wo der Puffer mit den Inhalten der MIDI Datei liegt und was/wie aus diesem Puffer verwendet wird.

Daten

Oft findet man auch nur eine Zahl wie `0x19870116`, was ganz klar nach einem Jahres Datum aussieht (Tag 16, 1 Monat (Januar), Jahr 1987). Das ist vielleicht das Geburtsdatum von jemandem (ein Programmierer. ihre/seine bekannte, Kind), oder ein anderes wichtiges Datum. Das Datum kann auch in umgekehrter folge auftreten, wie z.B `0x16011987`. Datums angaben im Amerikanischen-Stil sind auch weit verbreitet wie `0x01161987`.

Ein ziemlich bekanntes Beispiel ist `0x19540119` (magic number wird in der UFS2 Superblock Struktur benutzt), das Geburtsdatum von Marschall Kirk McKusick ist, einem Prominenten FreeBSD Entwickler.

Stuxnet benutzt die Zahl "19790509" (nicht als 32-Bit Zahl, aber als String), was zu Spekulationen geführt hat weil die malware Verbindungen nach Israel aufzeigt[20].

Solche Zahlen sind auch sehr beliebt in Amateur Kryptografie, zum Beispiel, ein Ausschnitt aus den *secret function* Interna aus dem HASP3 Dongle [21]:

```
void xor_pwd(void)
{
        int i;

        pwd^=0x09071966;
        for(i=0;i<8;i++)
        {
                al_buf[i]= pwd & 7; pwd = pwd >> 3;
        }
};

void emulate_func2(unsigned short seed)
{
        int i, j;
        for(i=0;i<8;i++)
        {
                ch[i] = 0;

                for(j=0;j<8;j++)
                {
```

[20] Das ist das Datum der Hinrichtung von Habib Elghanian, persischer Jude.

[21] https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm

```
                        seed *= 0x1989;
                        seed += 5;
                        ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
                }
        }
}
```

DHCP

Das Trifft auf Netzwerk Protokolle ebenso zu. Zum Beispiel, die Pakete des DHCP Protokoll's beinhalten so genannte *magic cookie*: 0x63538263. Jeder Code der ein DHCP Pakete generiert, muss diese Konstante in das Pakete einbetten. Wenn wir diesen Code finden, wissen wir auch wo es passiert und nicht nur was passiert. Jedes Programm das DHCP Pakete empfangen kann muss verifizieren das der *magic cookie* mit der Konstante übereinstimmt.

Zum Beispiel, lasst uns die dhcpcore.dll Datei aus Windows 7 x64 analysieren die nach der Konstante suchen. Wir können die Konstante zweimal finden: Es sieht danach aus als wäre die Konstante in zwei Funktionen benutzt mit dem selbst redenden Namen
DhcpExtractOptionsForValidation() und DhcpExtractFullOptions():

Listing 5.4: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ; DATA XREF:
    DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ; DATA XREF:
    DhcpExtractFullOptions+97
```

Und hier die (Speicher) Orte an denen auf die Konstante zugegriffen wird:

Listing 5.5: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

Und:

Listing 5.6: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

5.6.2 Spezifische Konstanten

Manchmal, gibt es spezifische Konstanten für gewissen Code Zum Beispiel, einmal hat der Autor sich in ein Stück Code gegraben wo die Nummer 12 verdächtig oft vor kam. Arrays haben oft eine Größe von 12 oder ein vielfaches von 12 (24, etc). Wie sich raus stellte, hat der Code eine 12-Kanal Audiodatei an der Eingabe entgegen genommen und sie verarbeitet.

Und umgekehrt: zum Beispiel, wenn ein Programm ein Textfeld verarbeitet das eine Länge von 120 Bytes hat, dann gibt es auch eine Konstante 120 oder 119 irgendwo im Code. Wenn UTF-16 Benutzt wird, dann $2 \cdot 120$. Wenn Code mit Netzwerkpaketen arbeitet die von fester Größe sind, ist es eine gute Idee nach dieser Konstante im Code zu suchen.

Das trifft auch auf Amateur Kryptografie zu (Lizenz Schlüssel, etc). Bei einem verschlüsselten Block von n Bytes, will man versuchen die vorkommen dieser Nummer im Code zu suchen, auch, wenn man ein Stück Code sieht der sich n mal während einer Schleifen Ausführung wiederholt, ist das vielleicht eine ver-/Entschlüsselung Routine.

5.6.3 Nach Konstanten suchen

Das ist einfach mit IDA: Alt-B oder Alt-I. Und für das suchen von Konstanten in einem Haufen großer Dateien, oder für das suchen in nicht ausführbaren Dateien, gibt es ein kleines Utility genannt *binary grep*[22].

5.7 Die richtigen Instruktionen finden

Wenn ein Programm auf die FPU Instruktionen zugreift und der Code selber enthält nur sehr wenige dieser Instruktionen, kann man diese einzeln mit einem Debugger überprüfen.

Zum Beispiel, eventuell haben wir Interesse daran wie Microsoft Excel die Formel berechnet die vom Benutzer eingegeben wurde. Zum Beispiel die Division Operation.

Wenn wir excel.exe (von Office 2010) in Version 14.0.4756.1000 in IDA laden ,ein komplettes Listig erstellen und jede FDIV Instruktion anschauen (ausgenommen die Instruktionen die eine Konstante als zweiten Parameter haben—diese Instruktionen interessieren uns nicht)

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

…dann sehen wir das es 144 FPU Instruktionen gibt.

Wir können einen String wie z.B =(1/3) in Excel eingeben und dann die Instruktionen überprüfen.

Beim prüfen jeder dieser Instruktionen in einem Debugger oder tracer (manche Prüfen 4 Instruktionen auf einmal), haben wir Glück und die gesuchte Instruktion ist die Nummer 14:

```
.text:3011E919 DC 33          fdiv    qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
```

[22] GitHub

```
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
FPU ST(0): 1.000000
```

ST(0) Beinhaltet das erste Argument (1) und das zweite Argument ist in [EBX].

Die Instruktion nach FDIV (FSTP) schreibt jedes Ergebnis in den Speicher:

```
.text:3011E91B DD 1E          fstp    qword ptr [esi]
```

Wenn wir einen Breakpoint auf diese Instruktion setzen können wir das Ergebnis betrachten:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Auch ein netter Scherz, wir können das Ergebnis auf die schnelle ändern:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel zeigt nun 666 in unserer Zelle, was uns letztendlich auch bestätigt das wir das richtige Ergebnis gefunden haben.

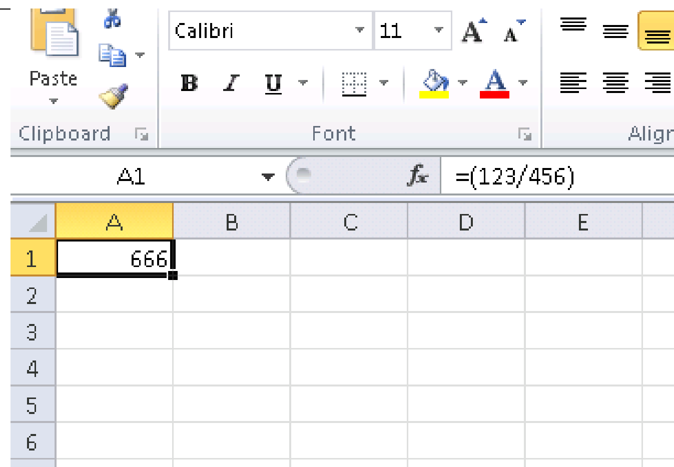


Abbildung 5.7: Der Scherz hat funktioniert

Wenn wir das gleiche mit der selben Excel Version versuchen, jedoch in 64-Bit Umgebungen. Dann finden wir nur noch 12 FDIV Instruktionen und die Instruktion nach der wir suchen ist die dritte.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

Es sieht danach aus als wären viele der Division Operationen der *float* und *double* Typen, vom Compiler mit SSE Instruktionen ersetzt wurden. Wie z.B DIVSD (DIVSD kommt insgesamt 268 mal vor).

5.8 Verdächtige Code muster

5.8.1 XOR Instruktionen

Instruktionen wie XOR op, op (zum Beispiel, XOR EAX, EAX) werden normal dafür benutzt Register Werte auf Null zu setzen, wenn jedoch einer der Operanden sich unterscheidet wird die „exclusive or" Operation ausgeführt.

Diese Operation wird allgemeinen selten benutzt beim programmieren, aber ist weit verbreitet in der Kryptografie, besonders bei Amateuren der Kryptografie. Sowas ist besonders Verdächtig wenn der zweite Operand eine große Zahl ist.

Das könnte ein Hinweis sein das etwas ver-/entschlüsselt wird oder Checksumme berechnet werden, etc.

Eine Ausnahme dieser Beobachtung ist der „canary" (1.20.3 on page 326). Die Generierung und das prüfen des „canary" werden oft mit Hilfe der XOR Instruktion gemacht.

Dieses AWK Skript kann benutzt werden um IDA listing (.lst) Dateien zu parsen:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="
    esp") if ($4!="ebp") { print $1, $2, tmp, ",", $4 } }' filename.lst
```

Es sollte auch noch erwähnt werden das diese Art von Skript in der Lage ist inkorrekt disassemblierten Code zu erkennen (**??** on page ??).

5.8.2 Hand geschriebener Assembler code

Moderne Compiler benutzen keine LOOP und RCL Instruktionen. Auf der anderen Seite sind diese Instruktionen sehr beliebt bei Programmieren die Code direkt in Assembler schreiben. Wenn man diese Instruktionen sieht, kann man mit hoher Sicherheit sagen das dieses Code Fragment händisch geschrieben wurde., Diese Instruktionen sind in der Instruktionsliste im Anhang mit (M) markiert: **??** on page ??.

Die Funktions Prolog und Epilog sind allgemein nicht vorhanden bei handgeschriebenen Assembler Code.

Tatsächlich gibt es kein bestimmtes System um Argumente an Funktionen zu übergeben wenn der Code handgeschrieben wurde.

Beispiel aus dem Windows 2003 Kernel (ntoskrnl.exe file):

```
MultiplyTest proc near               ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                          ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                       ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near               ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
             mov     eax, 417A000h
             mul     ecx
             cmp     edx, 2
             stc
             jnz     short locret_62057F
             cmp     eax, 0FE7A000h
             stc
             jnz     short locret_62057F
             clc
locret_62057F:                       ; CODE XREF: Multiply+10
                                     ; Multiply+18
             retn
Multiply     endp
```

Tatsächlich, wenn wir in den WRK[23] v1.2 source code schauen, kann dieser Code einfach in der Datei *WRK-v1.2\base\ntos\ke\i386\cpu.asm* gefunden werden.

[23] Windows Research Kernel

5.9 Using magic numbers while tracing

Oft ist unser Hauptziel zu verstehen wie ein Programm einen Wert behandelt der entweder über eine Datei oder über das Netzwerk erhalten wurde. Das manuelle tracen eines Wertes ist meistens ein ziemlich arbeits-intensiver Task. Eine der einfachsten Techniken um Werte zu Tracen (auch wenn nicht 100% verlässlich) ist eigene *magic number*'s zu benutzen.

Das ähnelt ein wenig dem Vorgang beim Röntgen auf gewisser weise: ein radioaktives Kontrastmittel wird dem Patienten injeziert, welches dann benutzt wird um die Gefässe des Patienten besser zu erkennen duch die Rönthgenstahlung. Wie das blut bei gesunden Menschen in den Nieren gereinigt wird wenn das Kontrastmittel im Blut ist, man kann dann sehr einfach auf dem Bild der Tomografie erkennen ob sich Nierensteine oder Tumore in den Nierenbefinden.

Wir können einfach eine 32-Bit Zahl nehmen z.B `0xbadf00d`, oder ein Geburtsdatum wie `0x11101979` und diese 4-Byte Zahl wird an einem bestimmten Punkt in eine Datei geschrieben welche von dem Programm das wir untersuchen genutzt wird.

Dann während das programm getraced wird mit tracer im *code coverage* modus, mit der Hilfe von *grep* oder durch einfaches durchsuchen der Textdatei (der trace Ergebnisse), können wir ganz einfach sehen wo der Wert benutzt wurde und wie er benutzt wurde.

Beispiel der *grepable* tracer Ergebnissen im *cc* mode:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0↙
    ↳ xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

Das gleiche verfahren kann man auch auf Netzwerkpakete anwenden. Für die *magic number* ist es wichtig das diese einzigartig ist und nicht im Programm code vorkommt.

Neben dem tracer Befehl, gibt es noch den DosBox (MS-DOS emulator) im heavydebug Modus, welcher in der Lage ist alle Informationen über alle Register zustände für jede ausgeführte Instruktion des Programmes in eine einfache Textdatei[24] zu schreiben, so kann diese Technik für DOS Programme nützlich sein.

5.10 Schleifen

Wann immer ein Programm mit einer Datei oder einem Puffer bestimmter Größe zu tun hat, muss dies eine Art von Verabeitungsschleife im code haben.

Dies ist ein reales Beispiel der tracer-Tool-Ausgabe, bei dem der Code auf irgendeine Weise codierte Datei von 258 Byte lud. Das Tool lief mit der Absicht die Zahl der An-

[24]See also my blog post about this DosBox feature: blog.yurichev.com

weisungen zukommen (ein DBI-Tool würde dies heutzutage sehr viel besser machen). Ich fand sehr schnell ein Code-Stück, welches 259/258 mal ausgeführt wurde.

```
...

0x45a6b5 e=   1 [FS: MOV [0], EAX] EAX=0x218fb08
0x45a6bb e=   1 [MOV [EBP-254h], ECX] ECX=0x218fbd8
0x45a6c1 e=   1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6c7 e=   1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e=   1 [JZ 45A9F2h] ZF=false
0x45a6d1 e=   1 [MOV [EBP-0Dh], 1]
0x45a6d5 e=   1 [XOR ECX, ECX] ECX=0x218fbd8
0x45a6d7 e=   1 [MOV [EBP-14h], CX] CX=0
0x45a6db e=   1 [MOV [EBP-18h], 0]
0x45a6e2 e=   1 [JMP 45A6EDh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0⤦
    ⤷ xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0⤦
    ⤷ x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0⤦
    ⤷ xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] ECX=0..5 (249 items skipped) 0xfe..0⤦
    ⤷ x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a6fb e= 258 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a701 e= 258 [MOV EAX, [EDX+10h]] [EDX+10h]=0x21ee4c8
0x45a704 e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0⤦
    ⤷ xfd..0x101
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0⤦
    ⤷ x102
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0⤦
    ⤷ xfd..0x101
0x45a710 e= 258 [MOVZX EAX, [EAX+EDX]] [EAX+EDX]=1..6 (156 items skipped) 0⤦
    ⤷ xf3, 0xf8, 0xf9, 0xfc, 0xfd
0x45a714 e= 258 [XOR EAX, ECX] EAX=1..6 (156 items skipped) 0xf3, 0xf8, 0⤦
    ⤷ xf9, 0xfc, 0xfd ECX=0x1f, 0x3e, 0x5d, 0x7c, 0x9b (248 items skipped) ⤦
    ⤷ 0x1ec2, 0x1ee1, 0x1f00, 0x1f1f, 0x1f3e
0x45a716 e= 258 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a71c e= 258 [MOV EDX, [ECX+10h]] [ECX+10h]=0x21ee4c8
0x45a71f e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0⤦
    ⤷ xfd..0x101
0x45a722 e= 258 [MOV [EDX+ECX], AL] AL=0..5 (77 items skipped) 0xe2, 0xee, ⤦
    ⤷ 0xef, 0xf7, 0xfc
0x45a725 e= 258 [JMP 45A6E4h]
0x45a727 e=   1 [PUSH 5]
0x45a729 e=   1 [MOV ECX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a72f e=   1 [CALL 45B500h]
0x45a734 e=   1 [MOV ECX, EAX] EAX=0x218fbd8
0x45a736 e=   1 [CALL 45B710h]
0x45a73b e=   1 [CMP EAX, 5] EAX=5
```

```
...
```

Wie sich herausstellte war dies auch eine Decodier-Schleife.

5.10.1 Muster in Binärdateien finden

Alle Beispiele hier wurden vorbereitet mit Windows mit aktiver Code Page 437 in der Konsole. Binär Dateien sehen intern etwas anders aus wenn eine andere Code page gesetzt ist.

Arrays

Manchmal kann man klar ein Array von 16/32/64-Bit Werten mit bloßem Auge im hex Editor erkennen.

Hier ist ein Beispiel eines 16-Bit Wertes. Wir sehen das das erste Byte ein paar aus 7 oder 8 ist und das zweite sieht zufällig aus:

```
E:\...3affacde09fe21c28f1543db51145b.dat h  1252        2175000      Col 0      23%    21:25
000007CA70: EF 07 C6 07 D6 07 26 08 | 0C 08 CE 07 24 07 60 07
000007CA80: CC 07 AA 07 A2 07 AC 07 | E9 07 BF 07 D6 07 2C 08
000007CA90: 09 08 CA 07 31 07 5E 07 | BC 07 9A 07 93 07 9E 07
000007CAA0: E6 07 BD 07 D8 07 2F 08 | 06 08 CB 07 3E 07 5E 07
000007CAB0: B3 07 91 07 8B 07 97 07 | E1 07 BB 07 DB 07 32 08
000007CAC0: 03 08 CB 07 4C 07 61 07 | AA 07 89 07 84 07 91 07
000007CAD0: E0 07 BB 07 DC 07 33 08 | 01 08 CC 07 57 07 64 07
000007CAE0: A4 07 84 07 81 07 90 07 | DE 07 BB 07 DE 07 34 08
000007CAF0: FF 07 CD 07 65 07 69 07 | A0 07 81 07 7F 07 90 07
000007CB00: DE 07 BC 07 DF 07 33 08 | FF 07 CE 07 70 07 6F 07
000007CB10: 9F 07 82 07 81 07 93 07 | DD 07 BC 07 E0 07 34 08
000007CB20: FE 07 CE 07 7E 07 78 07 | 9F 07 84 07 84 07 96 07
000007CB30: DE 07 BD 07 DF 07 32 08 | FF 07 CE 07 87 07 7F 07
000007CB40: A1 07 87 07 88 07 9B 07 | E2 07 BF 07 DE 07 2F 08
000007CB50: 02 08 CF 07 93 07 89 07 | A4 07 8C 07 8D 07 9F 07
000007CB60: E4 07 C0 07 DD 07 2D 08 | 03 08 CF 07 9C 07 92 07
000007CB70: A9 07 90 07 91 07 A3 07 | E6 07 C3 07 DD 07 2B 08
000007CB80: 04 08 D0 07 A7 07 9C 07 | AE 07 96 07 96 07 A7 07
000007CB90: E8 07 C7 07 DF 07 29 08 | 04 08 D3 07 B1 07 A7 07
000007CBA0: B4 07 9B 07 9B 07 AB 07 | E8 07 CA 07 E1 07 27 08
000007CBB0: 03 08 D5 07 BB 07 B3 07 | BB 07 A1 07 A0 07 AF 07
000007CBC0: EA 07 CD 07 E3 07 25 08 | 03 08 D8 07 C4 07 BD 07
000007CBD0: C1 07 A6 07 A5 07 B3 07 | EA 07 D1 07 E6 07 22 08
000007CBE0: 01 08 DC 07 CE 07 C8 07 | C8 07 AD 07 AA 07 B7 07
1Help  2Wrap  3Quit  4Text  5  6Edit  7Search  8OEM  9  10Quit
```

Abbildung 5.8: FAR: array von 16-Bit Werten

Ich habe eine Datei benutzt die ein 12 Kanal Signal digitalisiert mit 16-Bit nutzt ADC[25].

[25] Analog-to-Digital Converter

Und hier ist ein Beispiel von einem Typischen MIPS Code.

Wie wir uns vielleicht erinnern, jede MIPS (also auch ARM in ARM Mode oder ARM64) Instruktion hat eine Größe von 32 Bits (oder 4 Bytes), also ist solcher Code ein Array von 32-Bit Werten.

Wenn man den Screenshot anschaut, sehen wir eine Art Muster.

Vertikale und rote Linien wurden zur besseren Lesbarkeit eingefügt:

```
Hiew: FW96650A.bin
   FW96650A.bin                          FRO --------                    00005000
 00005000:  A0 B0 02 3C-04 00 BE AF-40 00 43 8C-21 F0 A0 03
 00005010:  FF 1F 02 3C-21 E8 C0 03-FF FF 42 34-24 10 62 00
 00005020:  00 A0 03 3C-25 10 43 00-04 00 BE 8F-08 00 E0 03
 00005030:  08 00 BD 27-F8 FF BD 27-A0 B0 02 3C-04 00 BE AF
 00005040:  48 00 43 8C-21 F0 A0 03-FF 1F 02 3C-21 E8 C0 03
 00005050:  FF FF 42 34-24 10 62 00-00 A0 03 3C-25 10 43 00
 00005060:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
 00005070:  21 10 00 00-04 00 BE AF-08 00 80 14-21 F0 A0 03
 00005080:  A0 B0 03 3C-21 E8 C0 03-44 29 02 7C-3C 00 62 AC
 00005090:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-01 00 03 24
 000050A0:  44 29 62 7C-A0 B0 03 3C-21 E8 C0 03-3C 00 62 AC
 000050B0:  04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27
 000050C0:  A0 B0 02 3C-04 00 BE AF-84 00 43 8C-21 F0 A0 03
 000050D0:  21 E8 C0 03-C4 FF 03 7C-84 00 43 AC-04 00 BE 8F
 000050E0:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
 000050F0:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-01 00 04 24
 00005100:  21 E8 C0 03-44 08 83 7C-20 00 43 AC-04 00 BE 8F
 00005110:  08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C
 00005120:  04 00 BE AF-20 00 43 8C-21 F0 A0 03-21 E8 C0 03
 00005130:  44 08 03 7C-20 00 43 AC-04 00 BE 8F-08 00 E0 03
 00005140:  08 00 BD 27-F8 FF BD 27-A0 B0 03 3C-04 00 BE AF
 00005150:  10 00 62 8C-01 00 08 24-04 A5 02 7D-08 00 09 24
 00005160:  10 00 62 AC-04 7B 22 7D-04 48 02 7C-04 84 02 7D
 00005170:  10 00 62 AC-21 F0 A0 03-21 18 00 00-A0 B0 0B 3C
 00005180:  51 00 0A 24-02 00 88 94-00 00 89 94-00 44 08 00
 00005190:  25 40 09 01-01 00 63 24-14 00 68 AD-F9 FF 6A 14
 000051A0:  04 00 84 24-21 18 00 00-A0 B0 0A 3C-07 00 09 24
 000051B0:  02 00 A4 94-00 00 A8 94-00 24 04 00-25 20 88 00
 1Global 2FilBlk 3CryBlk 4ReLoad 5       6String 7Direct 8Table  9       10Leave 11
```

Abbildung 5.9: Hiew: sehr typischer MIPS code

Ein weiteres Beispiel eines solchen Musters ist Buch: **??** on page ??.

Sparse Dateien

Diese dürftige Datei mit zerstreuten Daten inmitten einer fast leeren Datei. Jedes Space Zeichen hier ist in der tat ein Zero Byte (das wie ein space aussieht). Das ist eine Datei mit der ein FPGA Programmiert wird (Ein Altera Stratix GX Gerät). Sicher können Dateien wie diese einfach Komprimiert werden, aber diese Formate sind in der Wissenschaft und im Ingenieurs Wesen so wie in der Softwareentwicklung sehr verbreitet. Wo es oft um effizienten Zugriff geht und weniger um die Komprimierung der Daten.

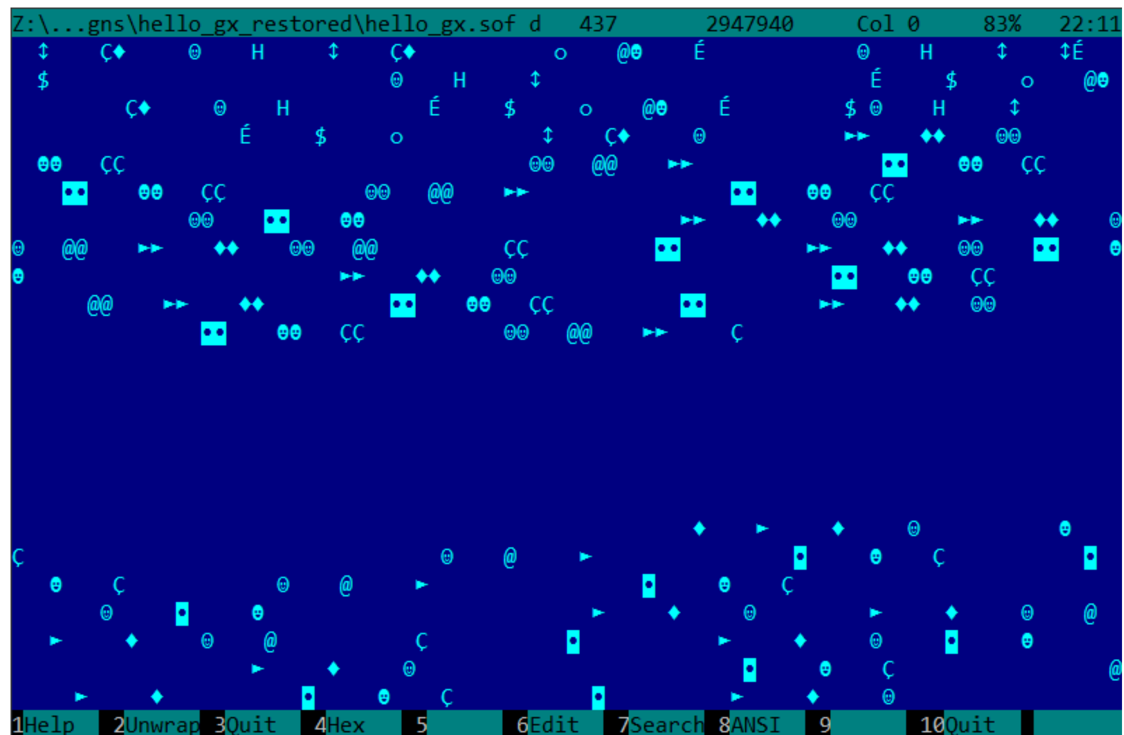


Abbildung 5.10: FAR: Sparse file

Komprimierte Dateien

Diese Datei ist einfach ein komprimiertes Archiv. Es hat eine relativ hohe Entropie und visuell betrachtet sieht es eher Chaotisch aus. So sehen komprimierte oder verschlüsselte Dateien aus.

```
Z:\...les\LISP_et_al\tinyscheme-1.41.zip d   437        67419     Col 0      67%    22:06
1Help  2Unwrap 3Quit  4Hex   5      6Edit  7Search 8ANSI  9      10Quit
```

Abbildung 5.11: FAR: Komprimierte Datei

CDFS[26]

BS Installationen werden üblicherweise als ISO Datei bereit gestellt, die Kopien von CD/DVD Disks sind. Das Dateisystem das benutzt wird heißt **cdfs!**[27], hier sieht man wie Dateinamen mit zusätzlichen Daten vermischt sind. Das können Datei Größen, Pointer auf andere Verzeichnisse, Datei Attribute und anderes sein. So sehen Dateisysteme typischerweise auch von innen aus.

```
Z:\...untu\ubuntu-15.10-desktop-i386.iso d    437     1226964992      Col 0       0%    21:59


                                                ä #       # ◘     ◘ s◙§►↕; ☻  ☺  ☺☺ SP•☺╜n
 PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕, s◙§►↕, s◙§►↕, CE└☺$       $        ф
  ф ` #       # ◘     ◘ s◙§►↕; ☻  ☺  ☺☺☺PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕, s◙
§►↕, s◙§►↕, n %       % ◘     ◘ s◙§►↕; ☻  ☺  ☺♣.DISKPX$☺mA     Am☺       ☺                TF
→☺♫s◙§►↕% s◙§►↕, s◙§►↕% NM◙☺ .diskn &       & ◘     ◘ s◙§►↕; ☻  ☺  ☺♦BOOT PX$☺mA     Am☺
   ☺                TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NMo☺ boot r (       ( ◘     ◘ s◙§►↕; ☻  ☺  ☺
♠CASPER PX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NM♂☺ casper n )
   ) ◘     ◘ s◙§►↕; ☻  ☺  ☺♣DISTSPX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s
◙§►↕→ NM◙☺ distsr 1       1 ◘     ◘ s◙§►↕; ☻  ☺  ☺•INSTALLPX$☺mA     Am☺       ☺
    TF→☺♫s◙§►↕( s◙§►↕, s◙§►↕( NM♀☺ installv 2       2 H     H s◙§►↕; ☻  ☺  ☺◘ISOLINUX PX$☺
mA     Am☺       ☺                TF→☺♫s◙§►↕( s◙§►↕( s◙§►↕( NM♪☺ isolinux z p☻     ☻p`♫
♫`s◙§►↕;     ☺  ☺◙MD5SUM.TXT PX$☺$ü     ü$☺       ☺                TF→☺♫s◙§►↕: s◙§►↕; s◙§►↕
: NM☼☺ md5sum.txt n ;       ; ◘     ◘ s◙§►↕; ☻  ☺  ☺♦PICS PX$☺mA     Am☺       ☺
    TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NMo☺ pics n <       < ◘     ◘ s◙§►↕; ☻  ☺  ☺♦POOL PX$☺mA
Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕   s◙§►↕→ NMo☺ pool r W       W ◘     ◘ s◙§►↕; ☻
  ☺  ☺•PRESEEDPX$☺mA     Am☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NM♀☺ preseed
è S☻     ☻Sα       αs◙§►↕;     ☺  ☺↕README.DISKDEFINES PX$☺$ü     ü$☺       ☺
TF→☺♫s◙§►↕→ s◙§►↕: s◙§►↕→ NM↨☺ README.diskdefines x ≤       ≤        s◙§►↕;     ☺  ☺•UBUNT
U.PX$☺mí     ím☺       ☺                TF→☺♫s◙§►↕→ s◙§►↕→ s◙§►↕→ NM♂☺ ubuntuSL•☺ ☻



1Help   2Unwrap 3Quit   4Hex    5       6Edit   7Search 8ANSI   9       10Quit
```

Abbildung 5.12: FAR: ISO file: Ubuntu 15 Installation CD[28]

[26] Compact Disc File System

[27] **cdfs!**

32-bit x86 ausführbarer Code

So sieht 32-Bit x86 ausführbarer Code aus. Der Code hat nicht wirklich viel Entropie, weil manche Bytes öfters vorkommen als andere.

Abbildung 5.13: FAR: Executable 32-bit x86 code

BMP graphics files

BMP Dateien sind nicht komprimiert, also ist jedes Byte (oder Gruppen von Bytes) beschrieben als ein Pixel. Diese Bild habe ich irgendwo in meiner Windows 8.1 Installation gefunden:



Abbildung 5.14: Example picture

Man kann sehen das dieses Bild Pixel hat, die nicht wirklich gut komprimiert werden könne (um das Zentrum herum), aber es sind lange ein-Farben Linien am Anfang und am ende der Datei. Tatsächlich Linien wie diese sehen wie Linien aus wenn man sich die Datei anschaut:

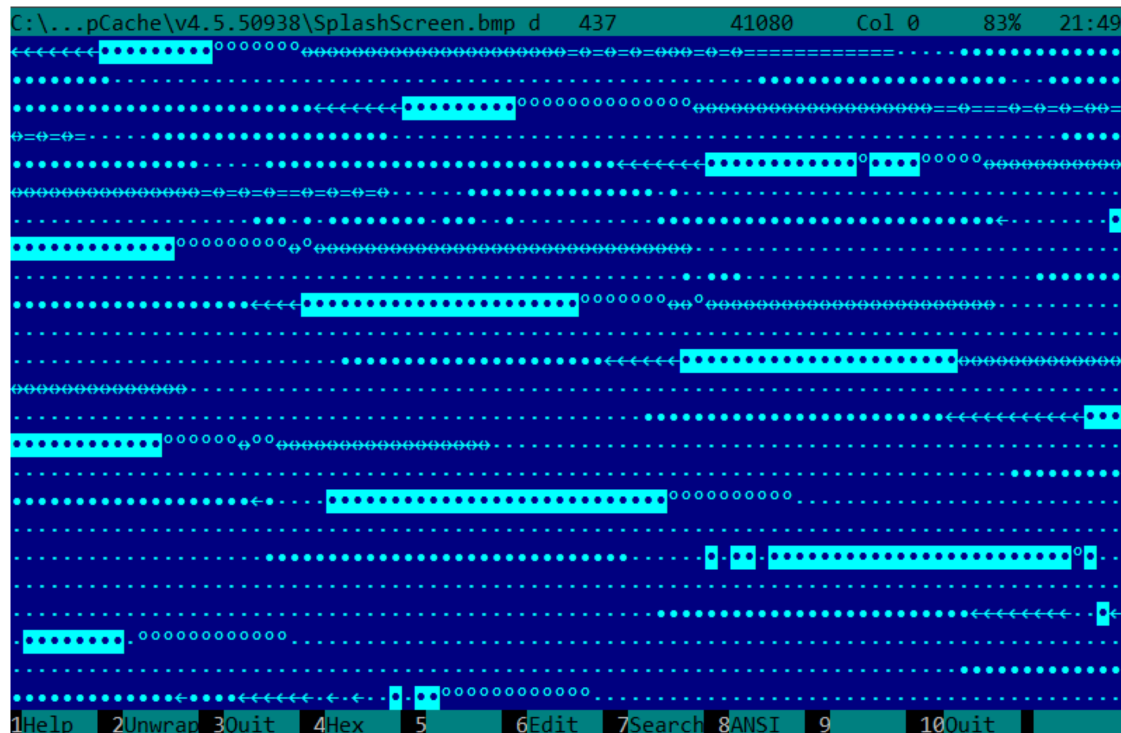


Abbildung 5.15: BMP file fragment

5.10.2 Memory „snapshots“ comparing

Die Technik zwei Memory Snapshots zu vergleichen ist recht einfach, das hat man auch oft benutzt um 8-Bit Computerspiele und „high score“'s zu hacken.

Zum Beispiel, wenn man ein geladenes Spiel auf einem 8-Bit Computer hat (auf den Maschinen ist nicht viel Speicher vorhanden, jedoch braucht das Spiel noch weniger Speicher) und du weißt was du im Spiel hast, sagen wir 100 Patronen, nun kann man einen „snapshot“ vom gesamten Speicher machen und diesen Irgendwohin speichern. Dann verschiesst man eine Patrone, dann geht der Patronen Zähler auf 99, nun erstellt man den zweiten Snapshot und Vergleich die beiden: Nun muss es irgendwo ein Byte geben das vorher 100 war und jetzt 99 ist.

Betrachtet man den Fakt das diese 8-Bit Spiele oftmals in Assembler geschrieben wurden und diese Variablen meist global waren, konnte man ziemlich einfach bestimmen welche Adressen im Speicher den Kugelzähler beinhalten. Wenn man nach allen Referenzen der Adresse im dissassembelten Spiel code sucht, ist es nicht schwer den Code decrementing zu finden und dann eine NOP Instruktion an diese Stelle zu schreiben, oder gar mehrere NOP-s, und dann hat man ein Spiel bei dem man für immer 100 Kugeln hat. Spiele auf 8-Bit Computern wurden allgemein an konstanten Adressen geladen, zusätzlich gab es nicht viele unterschiedliche Versionen des Spiels (Es war meist eine Version für lange Zeit populär), dadurch wussten enthusiastische Gamer welche Bytes (durch das benutzen von Basic Instruktionen wie

POKE) überschrieben werden mussten um das Spiel zu hacken. Das hat wiederum zu „cheat“ listen geführt die in Magazinen für 8-Bit Games erschienen, die dann POKE Instruktionen enthielten.

Es ist auch einfach „high score“ Dateien zu modifizieren, das funktioniert nicht nur bei 8-Bit Spielen. Man achte auf seinen Highscore Zähler, dann macht man ein Backup der Datei. Wenn sich der „high score“ Zähler ändert, vergleicht man die zwei Dateien miteinander, das kann man sogar mit dem DOS Tool FC[29] („high score“ Dateien, sind oft in Binärer Form).

Es wird beim Vergleichen der Dateien einen Punkt geben wo einige Bytes sich unterscheiden und es wird leicht sein, die Punkte zu sehen die die Bytes des Punktezähler beinhalten. Jedoch sind sich die Spiele Entwickler solcher Tricks bewusst und bauen Wege ein um das Programm vor solchen Manipulationen zu schützen.

Ein ähnliches Beispiel findet man auch in dem Buch **??** on page ??.

Windows registry

Es ist auch möglich die Windows Regestry zu vergleichen vor und nach der Programm Installation.

Es ist eine sehr populäre Methode Regestry Elemente zu finden die vom Programm benutzt werden. Vielleicht ist das auch der Grund warum die „windows regestry cleaner“ Shareware so populär ist.

Blink-comparator

Der Vergleich von Datei- oder Speichersnapshots erinnert ein wenig an einen Blinkkomparator [30] ein Gerät das in der Vergangenheit von Astronomen benutzt wurde, um sich bewegende Astronomische Objekte zu finden.

Ein Blinkkomperator erlaubt es schnell zwischen Photographie zu wechseln die zu unterschiedlicher Zeit aufgenommen wurden, so kann ein Astronom Unterschiede zwischen Fotografien visuell erkennen.

Ach übrigens, Pluto wurde durch einen solchen Blink-Komparator 1930 entdeckt.

5.11 Andere Dinge

5.11.1 Die Idee

Ein Reverse Engineer sollte versuchen so oft wie möglich in den Schuhen des Programmierers zu laufen. Um ihren/seinen Standpunkt zu betrachten uns sich selbst zu Fragen wie man einen Task in spezifischen Fällen lösen würde.

[29] MS-DOS Utility zum vergleichen von Dateien
[30] https://en.wikipedia.org/wiki/Blink_comparator

5.11.2 Anordnung von Funktionen in Binär Code

Sämtliche Funktionen die in einer einzelnen .c oder .cpp-Datei gefunden werden, werden zu den entsprechenden Objekt Dateien (.o) kompiliert. Später, fügt der Linker alle Objektdatein die er braucht zusammen, ohne die Reihenfolge oder die Funktionen in Ihnen zu verändern. Als eine Konsequenz, ergibt sich daraus wenn man zwei oder mehr aufeinander folgende Funktionen sieht, bedeutet dass das sie in der gleichen Source Code Datei platziert waren (Außer natürlich man bewegt sich an der Grenze zwischen zwei Dateien.). Das bedeutet das diese Funktionen etwas gemeinsam haben, das sie aus dem gleichen API-Level stammen oder aus der gleichen Library, etc.

5.11.3 kleine Funktionen

Sehr kleine oder leere Funktionen (1.3 on page 7) oder Funktionen die nur "true" (1) oder "false" (0) (**??** on page ??) sind weit verbreitet, und fast jeder ordentlicher Compiler tendiert dazu nur solche Funktionen in den resultierenden ausführbaren Code zu stecken, sogar wenn es mehrere gleiche Funktionen im Source Code bereits gibt. Also, wann immer man solche kleinen Funktionen sieht die z.B nur aus `mov eax, 1 / ret` bestehen und von mehreren Orten aus referenziert werden (und aufgerufen werden können), und scheinbar keine Verbindung zu einander haben, dann ist das wahrscheinlich das Ergebnis einer Optimierung.

5.11.4 C++

RTTI[31] (**??** on page ??)-data ist vielleicht auch nützlich für die C++ Klassen Identifikation.

[31] Run-Time Type Information

Kapitel 6

Betriebssystem-spezifische Themen

6.1 Methoden zur Argumentenübergabe (Aufrufkonventionen)

6.1.1 cdecl

Hierbei handelt es sich um die am weitesten verbreitete Methode um in C/C++-Sprachen Argumente an Funktionen zu übergeben.

Der caller muss den Wert des Stapel-Zeiger (ESP) auf den ursprünglichen Stand bringen, nachdem callee-Funktion beendet wurde.

Listing 6.1: cdecl

```
push Argument3
push Argument2
push Argument1
call Funktion
add esp, 12 ; gibt ESP zurueck
```

6.1.2 stdcall

Dies ist fast gleich zu der *cdecl*-Aufrufkonvention, mit Ausnahme, dass die callee den Wert von ESP auf den ursprünglichen Wert setzen muss. Dies geschieht durch die RET x anstatt RET, wobei gilt `x = Nummer des Arguments * sizeof(int)`[1].

Der caller passt den Stapel-Zeiger nicht an, es sind also keine `add esp, x`-Anweisungen vorhanden.

Listing 6.2: stdcall

[1] Die Größe einer Variablen vom Datentyp *int* ist 4 in x86-Systemen und 8 in x64-Systemen

```
push Argument3
push Argument2
push Argument1
call Funktion

Funktion:
;... tue etwas ...
ret 12
```

Diese Methode ist in Win32-Standard-Bibliotheken allgegenwärtig, fehlt jedoch in Win64 (siehe unten).

Beispielsweise kann die Funktion von 1.66 on page 108 genommen werden und durch Hinzufügen des `__stdcall`-Modifizierers leicht verändert werden:

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

Das Kompilat ist fast das gleiche wie bei 1.67 on page 108, jedoch wird RET 12 anstatt RET genutzt. Der **SP!** wird im caller nicht aktualisiert.

Als Konsequenz daraus kann die Anzahl der Funktionsargumente einfach von der RETN n-Anweisung abgeleitet werden, indem n durch 4 geteilt wird

Listing 6.3: MSVC 2010

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12
_f2@12  ENDP

; ...
        push    3
        push    2
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

Funktionen mit einer variablen Anzahl von Argumenten

`printf()`-ähnliche Funktionen sind vielleicht die einzigen Funktionen in C/C++ mit einer Variablen Anzahl von Argumenten, aber mit ihnen kann ein wichtiger Unterschied zwischen *cdecl* und *stdcall* veranschaulicht werden. Beginnen wir mit der Vorstellung, dass der Compiler die Anzahl der Argumente für jeden Aufruf von `printf()` kennt.

Die aufgerufene und bereits kompilierte `printf()`-Funktion befindet sich in der Datei MSVCRT.DLL (wenn über Windows geredet wird) und hat aber keinerlei Informationen darüber wie viele Argumente übergeben wurden; dies kann jedoch über den Formatstring herausgefunden werden.

Wenn `printf()` eine *stdcall*-Funktion wäre und den Stapel-Zeiger durch Zählen der Zahl der Argumente im Formatstring auf den ursprünglichen Wert setzen würde, kann eine gefährliche Situation entstehen, da ein Schreibfehler des Programmierers zu einem plötzliche Programmabsturz führen könnte. Aus diesem Grund ist für diese Art von Funktionen *stdcall* ungeeignet und *cdecl* ist zu bevorzugen.

6.1.3 fastcall

Dies ist der allgemeine Name für eine Methode, in der einige Argumente mittels Registern und der Rest über den Stack übergeben werden. Für ältere CPUs ist *fastcall* schneller als *cdecl* und *stdcall* wegen der geringeren Stack-Nutzung. Auf neueren CPUs wird dieser Ansatz vermutlich keine signifikante Geschwindigkeitserhöhung nach sich ziehen.

fastcall ist nicht standardisiert, so das verschiedene Compiler eine unterschiedliche Umsetzung machen können. Dies kann zu Problemen führen, wenn zwei DLLs genutzt werden, von denen eine die andere nutzt und durch das Nutzen verschiedener Compiler unterschiedliche *fastcall*-Aufrufkonventionen genutzt werden.

Sowohl MSVC als auch GCC übergeben das erste und zweite Argument über ECX und EDX und den Rest der Arguments mittels des Stacks.

Der Stapel-Zeiger muss vom callee auf den ursprünglichen Wert gesetzt werden, wie in *stdcall* auch.

Listing 6.4: fastcall

```
push Argument3
mov edx, Argument2
mov ecx, Argument1
call Funktion

Funktion:
.. tue etwas ..
ret 4
```

Beispielsweise kann die Funktion von 1.66 on page 108 genommen werden und durch Hinzufügen des `__fastcall`-Modifizierers leicht verändert werden:

```
int __fastcall f3 (int a, int b, int c)
{
```

```
        return a*b+c;
};
```

Nachfolgend das Kompilat:

Listing 6.5: Optimierender MSVC 2010 /Ob0

```
_c$ = 8         ; size = 4
@f3@12  PROC
; \_a\$ = ecx
; \_b\$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

Es ist erkennbar, dass der callee den **SP!** mit der RETN-Anweisung und einem Operanden auf den ursprünglichen Wert setzt.

Dies bedeutet, dass die Anzahl der Argumente ebenfalls einfach abgeleitet werden kann.

GCC regparm

Dies ist in gewisser Weise die Weiterentwicklung von *fastcall*[2]. Mit der `-mregparm`-Option ist es möglich festzulegen, wieviele Argumente per Register übergeben werden (maximal 3). Aus diesem Grund werden die Register EAX, EDX und ECX genutzt.

Natürlich werden, wenn die Anzahl der Argumente kleiner als drei ist, nicht alle drei Register genutzt.

Der caller setzt den Stapel-Zeiger auf den initialen Zustand.

Als Beispiel siehe (1.21.1 on page 357).

Watcom/OpenWatcom

Hier erfolgt der Aufruf mit der „Register-Aufruf-Konvention“. Die ersten vier Argumente werden in den Registern EAX, EDX, EBX und ECX übergeben, der Rest auf dem Stack.

[2] http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm

Diese Funktionen haben einen Unterstrich an den Funktionsnamen angehängt, um sie von den anderen Aufrufkonventionen unterscheiden zu können.

6.1.4 thiscall

Hier wird der *this*-Zeiger des Objekts an die Methode in C++ übergeben.

In MSVC wird *this* üblicherweise im ECX-Register übergeben.

In GCC wird der *this*-Zeiger im ersten Argument der Methode übergeben. Es ist sehr offensichtlich dass intern alle Methoden ein zusätzliches Argument haben.

Als Beispiel siehe (**??** on page ??).

6.1.5 x86-64

Windows x64

Die Art Argumente zu Übergeben ähnelt in Win64 in gewisser Weise `fastcall`. Die ersten vier Argumente werden in den Registern RCX, RDX, R8 und R9 übergeben und der Rest auf dem Stack. Der caller muss Platz für 32 Byte, also 4 64-Bit-Werte bereitstellen, so dass der callee dort die ersten vier Argumente speichern kann. Kurze Funktionen können die Werte der Argumente direkt aus den Registern lesen, während längere Funktionen diese für späteren Gebrauch zwischenspeichern sollten.

Der caller muss den Stapel-Zeiger auf den vorherigen Zustand zurücksetzen.

Diese Aufrufkonvention wird auch in den Windows x86-64-System-DLLs genutzt (anstatt *stdcall* in Win32).

Beispiel:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

Listing 6.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
```

```
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT:$SG2937
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP
```

Es ist hier klar erkennbar, wie sieben Argumente übergeben werden: vier in den Registern und die drei restlichen auf dem Stack.

Der Code des f1()-Funktionsprolog sichert die Argumente in dem „Scratch Space“, einer Stelle auf dem Stack, die genau für diese Zwecke existiert.

Dies ist so realisiert, weil der Compiler nicht sicher sein kann, dass genug Register ohne diese vier genutzt werden können und sie sonst durch die Argumente verändert werden können, bis die Funktion beendet wird.

Die Bereitstellung des „Scratch Space“ auf dem Stack ist Aufgabe der aufrufenden Funktion.

Listing 6.7: Optimierender MSVC 2012 /0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72
        ret     0
main    ENDP
```

Wenn das Beispiel ohne Optimierung compiliert wird, ist das Ergebnis fast das gleiche, lediglich der „Scratch Space“ ist unnötig und wird nicht genutzt.

Beachtenswert ist auch, wie MSVC 2012 das Laden von einfachen Werten in Register durch das Nutzen von LEA (**??** on page ??) optimiert. MOV wäre hier ein Byte länger (5 anstatt 4).

Ein weiteres Beispiel so eines Sachverhalts ist: **??** on page ??.

Windows x64: Übergeben von *this* (C/C++)

Der *this*-Zeiger wird in RCX übergeben, das erste Argument der Methode ist RDX, usw. Ein Beispiel ist hier zu sehen: **??** on page ??.

Linux x64

Die Art wie Linux für x86-64 Argumente übergibt ist fast die Gleiche wie in Windows, jedoch werden sechs anstatt vier Register genutzt (RDI, RSI, RDX, RCX, R8, R9) und es gibt keinen „Scratch Space“, auch wenn der callee die Registerwerte auf dem Stack speichern kann wenn er dies will oder muss.

Listing 6.8: Optimierender GCC 4.7.3

```
.LC0:
        .string "%d %d %d %d %d %d %d\n"
f1:
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main:
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

Zur Beachtung: die Werte werden hier in die 32-Bit-Teile der Register (z.B.EAX) geschrieben, aber nicht in die kompletten 64-Bit-Register (RAX). Dies wird gemacht, weil jeder Schreibzugang auf die niederwertigen 32-Bit-Teile eines Registers automatisch den höherwertigen Teil zurücksetzt.

Vermutlich wurde dies bei AMD so eingeführt um die Portierung des Codes zu x86-64 zu vereinfachen.

6.1.6 Rückgabewerte von *float*- und *double*-Typen

In allen Konventionen außer in Win64, werden di Werte vom Typ *float* oder *double* in dem FPU-Register `ST(0)` zurückgegeben.

In Win64 werden die Werte vom Typ *float* oder *double* in den niederwertigen 32 oder 64 Bit des `XMM0`-Registers zurückgegeben.

6.1.7 Verändern von Argumenten

Manchmal fragen C/C++-Programmierer (obwohl nicht auf diese PS beschränkt), was passieren kann wenn die Funktionsargumente verändert werden.

Die Antwort ist einfach: die Argumente sind auf dem Stack gespeichert und hier werden auch die Veränderungen vorgenommen.

Die aufzurufende Funktion wird diese nicht nach dem Verlassen des callee nutzen (der Autor dieser Linien hat so einen Fall in der Praxis noch nie gesehen).

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

Listing 6.9: MSVC 2012

```
_a$ = 8                                                 ; size = 4
_b$ = 12                                                ; size = 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

Also: ja, die Argumente können einfach modifiziert werden. Natürlich, wenn diese keine *Referenz* in C++ (**??** on page ??) ist, und nicht die Daten verändert werden auf die der Zeiger zeigt, wird der Effekt nicht außerhalb der aktuellen Funktion sichtbar sein.

Theoretisch kann der caller die modifizierten Argumente auf irgendeine Weise nutzen nachdem der callee beendet wurde. Vielleicht wenn dies direkt in Assembler programmiert ist.

Beispielsweise wird Code wie folgt von gewöhnlichen C/C++-Compilern erzeugt:

```
        push    456     ; wird gleich b sein
        push    123     ; wird gleich a sein
        call    f       ; f() modifiziert das erste Argument
        add     esp, 2*4
```

Der Code kann wie folgt neu geschrieben werden:

```
        push    456     ; wird gleich b sein
        push    123     ; wird gleich a sein
        call    f       ; f() modifiziert das erste Argument
        pop     eax
        add     esp, 4
        ; EAX=Erstes Argument von f() modifiziert in f()
```

Es ist scher vorzustellen warum jemand dies tun sollte, aber in der Praxis ist es möglich. Nichtsdestotrotz bietet der C/C++-Standard keinen Möglichkeit dies zu tun.

6.1.8 Einen Zeiger auf ein Argument verarbeiten

…mehr als das ist es sogar möglich, einen Zeiger auf ein Funktionsargument zu nehmen und an eine weitere Funktion zu übergeben:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
        modify_a (&a);
        printf ("%d\n", a);
};
```

Es ist schwierig die Funktionsweise zu verstehen, aber der folgende Code bring Klarheit:

Listing 6.10: Optimierender MSVC 2010

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
```

```
        lea     eax, DWORD PTR _a$[esp-4] ; just get the address of value in
    local stack
        push    eax                       ; and pass it to modify_a()
        call    _modify_a
        mov     ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
        push    ecx                       ; and pass it to printf()
        push    OFFSET $SG2796            ; '%d'
        call    _printf
        add     esp, 12
        ret     0
_f      ENDP
```

Die Adresse der Stelle im Stack an der a übergeben wird, wird lediglich an eine weitere Funktion übergeben. Diese verändert den Wert der mit dem Zeiger übergeben wird und `printf()` gibt anschließend den veränderten Wert aus.

Der aufmerksame Leser mag sich fragen, was mit der Aufrufkonvention ist, in der Funktionsargumente in Registern übergeben werden.

Das ist eine Situation, in der *Shadow Space* genutzt wird.

Der Eingangswert wird vom Register in den *Shadow Space* des lokalen Stacks kopiert und dann diese Adresse an die andere Funktion übergeben:

Listing 6.11: Optimierender MSVC 2012 x64

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
f       PROC
        mov     DWORD PTR [rsp+8], ecx   ; save input value in Shadow Space
        sub     rsp, 40
        lea     rcx, QWORD PTR a$[rsp]   ; get address of value and pass it
    to modify_a()
        call    modify_a
        mov     edx, DWORD PTR a$[rsp]   ; reload value from Shadow Space and
    pass it to printf()
        lea     rcx, OFFSET FLAT:$SG2994 ; '%d'
        call    printf
        add     rsp, 40
        ret     0
f       ENDP
```

GCC sichert den Eingangswert ebenfalls auf dem lokalen Stack:

Listing 6.12: Optimierender GCC 4.9.1 x64

```
.LC0:
        .string "%d\n"
f:
        sub     rsp, 24
        mov     DWORD PTR [rsp+12], edi  ; store input value to the local
    stack
        lea     rdi, [rsp+12]            ; take an address of the value and
    pass it to modify_a()
        call    modify_a
```

```
        mov     edx, DWORD PTR [rsp+12]  ; reload value from the local stack
    and pass it to printf()
        mov     esi, OFFSET FLAT:.LC0    ; '%d'
        mov     edi, 1
        xor     eax, eax
        call    __printf_chk
        add     rsp, 24
        ret
```

GCC für ARM64 tut genau das gleiche, allerdings wird hier der Platz *Register Save Area* genannt:

Listing 6.13: Optimierender GCC 4.9.1 ARM64

```
f:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0          ; setup FP
        add     x1, x29, 32         ; calculate address of variable in
    Register Save Area
        str     w0, [x1,-4]!        ; store input value there
        mov     x0, x1              ; pass address of variable to the
    modify_a()
        bl      modify_a
        ldr     w1, [x29,28]        ; load value from the variable and pass it
    to printf()
        adrp    x0, .LC0            ; '%d'
        add     x0, x0, :lo12:.LC0
        bl      printf              ; call printf()
        ldp     x29, x30, [sp], 32
        ret
.LC0:
        .string "%d\n"
```

Übrigens eine gleiche Nutzung des *Shadow Space* wird auch hier beschrieben: **??** on page ??.

6.2 lokaler Thread-Speicher

TLS ist ein Datenbereich der für jeden einzelnen Thread spezifisch ist. Jeder Thread kann hier alles speichern was er möchte. Eine bekanntes Beispiel ist die globale C-Standardvariable *errno*.

Mehrere Threads können Funktionen simultan aufrufen, die einen Fehlercode in *errno* zurückgeben. Eine globale Variable würde hier nicht korrekt für Multithread-Programme funktionieren, aus diesem Grund muss *errno* im TLS gesichert.

Im C++11-Standard wurde ein neues Schlüsselwort *thread_local* eingeführt, um anzuzeigen, dass jeder Thread eine eigene Kopie dieser Variable, die initialisiert werde kann und sich auf dem TLS befindet[3]:

[3] C11 hat ebenfalls einen (optionalen) Thread-Support

Listing 6.14: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

Kompiliert mit MinGW GCC 4.8.1 jedoch nicht MSVC 2012.

Wenn es um PE-Dateien geht, also die ausführbaren Dateien, wird die *tmp*-Variable in der Sektion mit dem Namen TLS gesichert.

6.2.1 Nochmals Linearer Kongruenzgenerator

Der Pseudozufallszahlen-Generator der in 1.22 on page 397 bereits erwähnt wurde hat einen Nachteil: Er ist nicht Thread-sicher, weil er eine interne Zustandsvariable hat, die von verschiedenen Threads gleichzeitig gelesen und verändert werden kann.

Win32

Uninitialisierte TLS-Daten

Eine mögliche Lösung ist es den `__declspec( thread )`-Modifizierer zu der globalen Variable hinzuzufügen, so dass diese im TLS alloziert wird (Zeile 9):

```
1  #include <stdint.h>
2  #include <windows.h>
3  #include <winnt.h>
4
5  // from the Numerical Recipes book:
6  #define RNG_a 1664525
7  #define RNG_c 1013904223
8
9  __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13         rand_state=init;
14 }
15
16 int my_rand ()
17 {
18         rand_state=rand_state*RNG_a;
19         rand_state=rand_state+RNG_c;
20         return rand_state & 0x7fff;
21 }
22
```

```
int main()
{
        my_srand(0x12345678);
        printf ("%d\n", my_rand());
};
```

Hiew zeigt, dass eine neue PE-Sektion in der ausführbaren Datei existiert: `.tls`.

Listing 6.15: Optimierender MSVC 2013 x86

```
_TLS    SEGMENT
_rand_state DD  01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG84851 DB     '%d', 0aH, 00H
_DATA   ENDS
_TEXT   SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state[ecx], eax
        ret     0
_my_srand ENDP

_my_rand PROC
; FS:0=address of TIB
        mov     eax, DWORD PTR fs:__tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
        mov     ecx, DWORD PTR __tls_index
        mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
        imul    eax, DWORD PTR _rand_state[ecx], 1664525
        add     eax, 1013904223          ; 3c6ef35fH
        mov     DWORD PTR _rand_state[ecx], eax
        and     eax, 32767               ; 00007fffH
        ret     0
_my_rand ENDP

_TEXT   ENDS
```

`rand_state` befindet sich nun im TLS-Segment und jeder Thread hat seine eigene Kopie dieser Variable.

Auf diese Variable wird wie folgt zugegriffen: lade die Adresse von TIB von FS:2Ch, anschließend addiere einen zusätzlichen Index (falls notwendig), zuletzt berechne die Adresse des TLS-Segments.

Nun ist es möglich auf die rand_state-Variable über des ECX-Register zuzugreifen, welches auf eine spezifische Stelle in jedem Thread zeigt.

Der FS:-Selektor ist bei jedem Reverse Engineer gut bekannt und zeigt immer auf TIB, so dass es schnell möglich ist thread-spezifische Daten zu laden.

Der GS:-Selektor wird unter Win64 genutzt, die Adresse von TLS ist 0x58:

Listing 6.16: Optimierender MSVC 2013 x64

```
_TLS    SEGMENT
rand_state DD   01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG85451 DB     '%d', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT

init$ = 8
my_srand PROC
        mov     edx, DWORD PTR _tls_index
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     r8d, OFFSET FLAT:rand_state
        mov     rax, QWORD PTR [rax+rdx*8]
        mov     DWORD PTR [r8+rax], ecx
        ret     0
my_srand ENDP

my_rand PROC
        mov     rax, QWORD PTR gs:88 ; 58h
        mov     ecx, DWORD PTR _tls_index
        mov     edx, OFFSET FLAT:rand_state
        mov     rcx, QWORD PTR [rax+rcx*8]
        imul    eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH
        add     eax, 1013904223         ; 3c6ef35fH
        mov     DWORD PTR [rcx+rdx], eax
        and     eax, 32767              ; 00007fffH
        ret     0
my_rand ENDP

_TEXT   ENDS
```

Initialisierte TLS-Daten

Angenommen es soll ein fester Wert für rand_state gesetzt werden und der Programmierer vergisst dies, wird dies automatisch in Zeile 9 gemacht:

```
1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
```

```
// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state=1234;

void my_srand (uint32_t init)
{
        rand_state=init;
}

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        printf ("%d\n", my_rand());
};
```

Der Code ist nicht anders als zuvor, aber in IDA sieht man folgendes:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls            segment para public 'DATA' use32
.tls:00404000                 assume cs:_tls
.tls:00404000                 ;org 404000h
.tls:00404000 TlsStart        db    0         ; DATA XREF:
    .rdata:TlsDirectory
.tls:00404001                 db    0
.tls:00404002                 db    0
.tls:00404003                 db    0
.tls:00404004                 dd 1234
.tls:00404008 TlsEnd          db    0         ; DATA XREF: .rdata:TlsEnd_ptr
...
```

Jedes mal wenn ein neuer Thread gestartet wird, wird ein neuer TLS alloziert und alle Daten, inklusive der 1234 dorthin kopiert.

Dies ist eine typische Situation:

- Thread A wird gestartet und ein TLS wird dafür erstellt. 1234 wird in `rand_state` kopiert.
- Die Funktion `my_rand()` wird einige Male in Thread A aufgerufen. `rand_state` unterscheidet sich 1234.
- Thread B wird gestartet und ein TLS wird dafür erstellt. 1234 wird in `rand_state` kopiert, während Thread A einen anderen Wert in der gleichen Variable hat.

TLS-Callback-Funktionen

Was passiert wenn die Variablen in TLS mit Daten gefüllt werden, die in einer bestimmten Weise verändert werden sollen?

Angenommen es existiert folgende Aufgabe: Der Programmierer hat vergessen die my_srand()-Funktion aufzurufen um den PRNG zu initialisieren, dieser muss jedoch initialisiert werden um „echte" Zufallszahlen anstatt den Wert 1234 zu erzeugen. In diesem Fall kann die TLS-Callback-Funktion genutzt werden.

Der folgende Code ist nicht sehr portabl, nichtsdestotrotz ist die Idee dahinter erkennbar.

Was hier passiert ist eine Funktion zu definieren (tls_callback()) welche *vor* dem Starten eines Prozesses oder Threads aufgerufen wird.

Die Funktion initialisiert den PRNG mit dem Wert der von GetTickCount() zurückgegeben wird.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
        rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
        my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
}

int main()
{
        // rand_state is already initialized at the moment (using
    GetTickCount())
```

```
        printf ("%d\n", my_rand());
};
```

Nachfolgend das Ergebnis in IDA:

Listing 6.17: Optimierender MSVC 2013

```
.text:00401020 TlsCallback_0   proc near               ; DATA XREF:
    .rdata:TlsCallbacks
.text:00401020                 call    ds:GetTickCount
.text:00401026                 push    eax
.text:00401027                 call    my_srand
.text:0040102C                 pop     ecx
.text:0040102D                 retn    0Ch
.text:0040102D TlsCallback_0   endp

...

.rdata:004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF:
    .rdata:TlsCallbacks_ptr

...

.rdata:00402118 TlsDirectory    dd offset TlsStart
.rdata:0040211C TlsEnd_ptr      dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr    dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS-Callback-Funktionen werden manchmal genutzt um Routinen zu entpacken und deren Funktion zu verschleiern.

Manche Menschen sind verwirrt darüber, dass einiger Code genau vor dem OEP[4] ausgeführt wird.

Linux

Nachfolgend ein Beispiel wie eine thread-lokale, globale Variable in GCC deklariert wird:

```
__thread uint32_t rand_state=1234;
```

Dies is ein Standard-C/C++-Modifizierer sondern eine GCC-spezifische Erweiterung[5].

Der GS:-Selektor wird ebenso genutzt um auf den TLS-Bereich zuzugreifen, jedoch in einer etwas anderen Art:

Listing 6.18: Optimierender GCC 4.8.1 x86

```
.text:08048460 my_srand        proc near
.text:08048460
```

[4] Original Entry Point
[5] https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/C99-Thread-Local-Edits.html

```
.text:08048460 arg_0           = dword ptr  4
.text:08048460
.text:08048460                 mov     eax, [esp+arg_0]
.text:08048464                 mov     gs:0FFFFFFFCh, eax
.text:0804846A                 retn
.text:0804846A my_srand        endp

.text:08048470 my_rand         proc near
.text:08048470                 imul    eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B                 add     eax, 3C6EF35Fh
.text:08048480                 mov     gs:0FFFFFFFCh, eax
.text:08048486                 and     eax, 7FFFh
.text:0804848B                 retn
.text:0804848B my_rand         endp
```

Mehr darüber in [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)][6].

6.3 Systemaufrufe

Wie bekannt werden alle laufende Prozesse in einem BS in zwei Kategorien unterteilt: diejenigen, die vollen Zugriff auf die Hardware haben („kernel space“) und diejenigen die dies nicht haben („user space“).

Der Kernel des Betriebssystems sowie die Treiber gehören in der Regel zur ersten Kategorie.

Alle Anwendungen gehören in der Regel der zweiten Kategorie an.

Beispielsweise gehört der Linux-Kernel in den *kernel space* während Glibc im *user space* ausgeführt wird.

Diese Unterscheidung ist von großer Bedeutung für die Sicherheit eines BS: es ist sehr wichtig nicht jedem Prozess die Möglichkeit etwas in einem anderen Prozess oder sogar im BS-Kernel zum Absturz zu bringen. Auf der anderen Seite führt ein Fehler im Treiber oder innerhalb des BS-Kernels in der Regel zu einem Kernel-Panic oder BSOD[7].

Der Schutz im x86-Prozessor erlaubt die Unterscheidung in vier unterschiedliche Schutzlevel (Ringe). Sowohl von Linux als auch von Windows werden jedoch nur zwei genutzt: Ring 0 („kernel space“) und Ring 3 („user space“).

Systemaufrufe (syscalls) sind die Stelle an der diese beiden Bereiche miteinander verbunden werden.

In diesem Sinne sind die Systemaufrufe die Haupt-API für die Anwendungen.

Unter Windows NT, befindet sich die Tabelle mit Systemaufrufen in der SSDT[8].

Die Nutzung von Systemaufrufen ist sehr verbreitet bei Shellcode- und Viren-Programmierern, weil es schwieriger ist die Adresse einer benötigten Funktion herauszufinden als ei-

[6]Auch verfügbar als `http://www.akkadia.org/drepper/tls.pdf`
[7]Blue Screen of Death
[8]System Service Dispatch Table

nen Systemaufruf zu nutzen. Die Kehrseite ist, das viel mehr Code geschrieben werden muss, aufgrund dem geringeren Grad an Abstraktion der API.

Erwähnenswert ist es das die Anzahl der Systemaufrufe bei den unterschiedlichen Betriebssystemen variieren kann.

6.3.1 Linux

Unter Linux wird ein Systemaufruf in der Regel mit `int 0x80` aufgerufen. Die Nummer des Aufrufs wird im EAX-Register übergeben und die restlichen Parameter in anderen Registern.

Listing 6.19: Ein einfaches Beispiel zur Nutzung zweier Systemaufrufe

```
section .text
global  _start

_start:
        mov     edx,len ; buffer len
        mov     ecx,msg ; buffer
        mov     ebx,1   ; file descriptor. 1 is for stdout
        mov     eax,4   ; syscall number. 4 is for sys_write
        int     0x80

        mov     eax,1   ; syscall number. 1 is for sys_exit
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Kompilation:

```
nasm -f elf32 1.s
ld 1.o
```

Eine vollständige Liste von Systemaufrufen unter Linux: `http://syscalls.kernelgrok.com/`.

Um Systemaufrufe unter Linux zu unterbrechen und nachverfolgen zu können, kann strace(7.2.3 on page 657) genutzt werden.

6.3.2 Windows

Hier werden die Systemaufrufe via `int 0x2e` aufgerufen oder über die spezielle x86-Anweisung SYSENTER.

Eine vollständige Liste von Systemaufrufen unter Windows: `http://j00ru.vexillium.org/ntapi/`.

Weitere Informationen:

„Windows Syscall Shellcode“ von Piotr Bania: http://www.symantec.com/connect/articles/windows-syscall-shellcode.

6.4 Linux

6.4.1 Positionsabhängiger Code

Wenn der Code von Shared Libraries (.so) unter Linux analysiert wird, findet man häufig das folgende Code-Muster:

Listing 6.20: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near    ; CODE XREF: sub_17350+3
.text:0012D5E3                                    ; sub_173CC+4 ...
.text:0012D5E3       mov     ebx, [esp+0]
.text:0012D5E6       retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near          ; CODE XREF: tmpfile+73

...

.text:000576C0       push    ebp
.text:000576C1       mov     ecx, large gs:0
.text:000576C8       push    edi
.text:000576C9       push    esi
.text:000576CA       push    ebx
.text:000576CB       call    __x86_get_pc_thunk_bx
.text:000576D0       add     ebx, 157930h
.text:000576D6       sub     esp, 9Ch

...

.text:000579F0       lea     eax, (a__gen_tempname - 1AF000h)[ebx] ;
    "__gen_tempname"
.text:000579F6       mov     [esp+0ACh+var_A0], eax
.text:000579FA       lea     eax, (a__SysdepsPosix - 1AF000h)[ebx] ;
    "../sysdeps/posix/tempname.c"
.text:00057A00       mov     [esp+0ACh+var_A8], eax
.text:00057A04       lea     eax, (aInvalidKindIn_ - 1AF000h)[ebx] ;
    "! \"invalid KIND in __gen_tempname\""
.text:00057A0A       mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12       mov     [esp+0ACh+var_AC], eax
.text:00057A15       call    __assert_fail
```

Alle Zeiger auf Zeichenketten sind durch Konstanten und den Wert in EBX korrigiert, welcher zu Beginn jeder Funktion berechnet wird.

Dies ist sogenannter PIC[9], und hat den Zweck an jeder beliebigen Stelle im Speicher

[9] Position Independent Code

ausführbar zu sein. Aus diesem Grund können keine absoluten Speicheradressen verwendet werden.

PIC war entscheidend in früheren Computer-Systemen und ist es immer noch in Eingebetteten Systeme ohne virtuelle Speicherverwaltung in denen sich alle Prozesse in einem einzigen durchgängigen Speicherbereich befinden.

Diese Technik wird auch heute noch in *NIX-Systemen für Shared Libraries verwendet da diese von mehreren Prozessen genutzt, aber nur einmal in den Speicher geladen werden. Jeder Prozess kann jedoch die gleiche Bibliothek an verschiedene Adressen „mappen“. Aus diesem Grund muss diese Bibliothek auch ohne Verwendung absoluter Adressen funktionieren.

Machen wir ein sehr einfaches Experiment:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Nachfolgende die kompilierte .so-Datei von GCC 4.7.3. in IDA:

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 6.21: GCC 4.7.3

```
.text:00000440                 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near         ;
    CODE XREF: _init_proc+4
.text:00000440                                         ;
    deregister_tm_clones+4 ...
.text:00000440                 mov     ebx, [esp+0]
.text:00000443                 retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                 public f1
.text:00000570 f1              proc near
.text:00000570
.text:00000570 var_1C          = dword ptr -1Ch
.text:00000570 var_18          = dword ptr -18h
.text:00000570 var_14          = dword ptr -14h
.text:00000570 var_8           = dword ptr -8
.text:00000570 var_4           = dword ptr -4
.text:00000570 arg_0           = dword ptr  4
.text:00000570
.text:00000570                 sub     esp, 1Ch
.text:00000573                 mov     [esp+1Ch+var_8], ebx
.text:00000577                 call    __x86_get_pc_thunk_bx
.text:0000057C                 add     ebx, 1A84h
```

```
.text:00000582                  mov     [esp+1Ch+var_4], esi
.text:00000586                  mov     eax, ds:(global_variable_ptr - 2000h↙
    ↘ )[ebx]
.text:0000058C                  mov     esi, [eax]
.text:0000058E                  lea     eax, (aReturningD - 2000h)[ebx] ;
    "returning %d\n"
.text:00000594                  add     esi, [esp+1Ch+arg_0]
.text:00000598                  mov     [esp+1Ch+var_18], eax
.text:0000059C                  mov     [esp+1Ch+var_1C], 1
.text:000005A3                  mov     [esp+1Ch+var_14], esi
.text:000005A7                  call    ___printf_chk
.text:000005AC                  mov     eax, esi
.text:000005AE                  mov     ebx, [esp+1Ch+var_8]
.text:000005B2                  mov     esi, [esp+1Ch+var_4]
.text:000005B6                  add     esp, 1Ch
.text:000005B9                  retn
.text:000005B9 f1               endp
```

Das ist es: die Zeiger auf *«returning %d\n»* und *global_variable* werden bei jedem Funktionsaufruf korrigiert.

Die `__x86_get_pc_thunk_bx()`-Funktion gibt in EBX die Adresse auf eine Stelle nach einen Aufruf von sich selbst zurück (hier `0x57C`).

Dies ist eine einfache Möglichkeit um den Wert des Programmzählers (EIP) an einer beliebigen Stelle zu erhalten. Die Konstante `0x1A84` gehört zu dem Unterschied des Funktionsanfangs und der sogenannten *Global Offset Table Procedure Linkage Table* (GOT PLT), die Sektion direkt hinter der *Global Offset Table* (GOT), an der der Zeiger auf *global_variable*. IDA zeigt diese Offsets aus Gründen des einfacheren Verständnisses in einer verarbeiteten Form an, der Code ist aber wie folgt:

```
.text:00000577                  call    __x86_get_pc_thunk_bx
.text:0000057C                  add     ebx, 1A84h
.text:00000582                  mov     [esp+1Ch+var_4], esi
.text:00000586                  mov     eax, [ebx-0Ch]
.text:0000058C                  mov     esi, [eax]
.text:0000058E                  lea     eax, [ebx-1A30h]
```

Hier zeigt EBX auf die `GOT` PLT-Sektion und um den Zeiger auf *global_variable* zu berechnen (welcher in der `GOT` gesichert ist) muss `0xC` subtrahiert werden.

Um den Zeiger auf die Zeichenkette *«returning %d\n»* zu berechnen muss `0x1A30` abgezogen werden.

Übrigens ist dies der Grund warum die AMD64-Anweisungen RIP[10]-relative Adressierung unterstützen: sie vereinfachen den PIC-Code.

Nachfolgend der selbe C-Code mit der gleichen GCC-Version, jedoch für x64 kompiliert.

IDA würde den resultierenden Code vereinfachen aber auch die Details zur RIP-relativen Adressierung unterdrücken. Um alles sehen zu können wird hier *objdump* anstatt IDA genutzt.

[10] Programmzähler in AMD64

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0x2008b9]        ↙
    ↳ ;
    200fe0 <_DYNAMIC+0x1d0>
 727:   53                      push   rbx
 728:   89 fb                   mov    ebx,edi
 72a:   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        ;
    751 <_fini+0x9>
 731:   bf 01 00 00 00          mov    edi,0x1
 736:   03 18                   add    ebx,DWORD PTR [rax]
 738:   31 c0                   xor    eax,eax
 73a:   89 da                   mov    edx,ebx
 73c:   e8 df fe ff ff          call   620 <__printf_chk@plt>
 741:   89 d8                   mov    eax,ebx
 743:   5b                      pop    rbx
 744:   c3                      ret
```

0x2008b9 ist der Unterschied zwischen der Adresse der Anweisung an 0x720 und *global_variable*, und 0x20 ist der Unterschied zwischen der Adresse der Anweisung an 0x72A und der Zeichenkette *«returning %d\n»*.

Wie zu sehen ist, die Notwendigkeit die Adressen regelmäßig neu zu berechnen macht die Ausführung etwas langsamer (auch wenn dies in x64 etwas besser ist).

Es mag also von Vorteil sein, statisch zu linken wenn Geschwindigkeit eine Rolle spielt [siehe: Agner Fog, *Optimizing software in C++* (2015)].

Windows

Der PIC-Mechanismus wird in Windows-DLLs nicht genutzt. Wenn der Windows-Loader eine DLL an eine andere Basisadresse laden muss, „patched“ er diese im Speicher (and den *FIXUP*-Platz) um alle Adressen zu korrigieren.

Dies bedeutet, dass mehrere Windows-Prozesse eine einmal geladene DLL nicht teilen können wenn diese an verschiedenen Adressen in verschiedenen Prozess-Speichern sein muss, da jede Instanz im Speicher die Funktionen an einer festen Adresse erwartet.

6.4.2 *LD_PRELOAD*-Hack in Linux

Diese Technik erlaubt es eigene, dynamische Bibliotheken vor anderen zu laden… sogar vor denen des Systems, wie libc.so.6.

Dies wiederum erlaubt es die eigenen Funktionen für die des Systems zu „ersetzen“. Es ist zum Beispiel einfach alle Aufrufe zu time(), read(), write(), usw. abzufangen.

Sehen wir uns einmal an, wie das Tool *uptime* ausgetrickst werden kann. Wie bekannt ist, zeigt dieses Programm an, wie lange der Computer schon arbeitet. Mithilfe von strace(7.2.3 on page 657), ist es möglich zu sehen, dass das Tool die Informationen aus der Datei /proc/uptime ausliest:

```
$ strace uptime
```

```
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

Es handelt sich dabei nicht um eine reale Datei auf der Festplatte sondern um eine virtuelle, bei der die Dateien on-the-fly im Linux Kernel erstellt werden. Es gibt zwei Zahlen:

```
$ cat /proc/uptime
416690.91 415152.03
```

Nachfolgend, der englischen Wikipedia[11]:

> The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

Versuchen wir eine eigene dynamische Bibliothek mit den Funktionen open(), read() und close() zu schreiben die so funktionieren wie wir es gerne hätten.

Zunächst wird unsere open()-Funktion den Namen der zu öffnenden Datei mit dem was wir brauchen vergleichen. Ist dies der Fall, soll der Deskriptor der geöffnete Datei geschrieben werden.

Als zweites read(): Wenn diese Funktion für den Datei-Deskriptor aufgerufen wird, soll die Ausgaben ersetzt werden und der Rest dem original read() aus libc.so.6 entsprechen. close() wird eine Meldung geben wenn die Datei der zur Zeit gefolgt wird geschlossen wurde.

Wir werden die dlopen()- und dlsym()-Funktionen nutzen, um die Adressen der Original-Funktionen in libc.so.6 herauszufinden.

Diese werden benötigt, weil die Ausführkontrolle wieder an die „realen" Funktionen übergeben werden müssen.

Auf der anderen Seite: wenn wir strcmp() unterbrechen und jeden einzelnen Vergleich von Zeichenketten im Programm untersuchen, müssten wir eine eigene strcmp()-Variante schreiben und nicht die Original-Funktion nutzen[12], was einfacher wäre.

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
```

[11] https://en.wikipedia.org/wiki/Uptime

[12] Als Beispiel, wie einfach strcmp()-Unterbrechung funktioniert [13] von Yong Huang

```
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;

        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its file descriptor
        else
                opened_fd=0;
        return fd;
};
```

```
int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff, 0↙
    ↳ x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

(Quellcode)

Kompilieren wir den Code als gemeinsame, dynamische Bibliothek:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Jetzt starten wir *uptime* während unsere Bibliothek vor den anderen geladen wird:

```
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Und wir sehen:

```
 01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, 0.01, 0.05
```

Wenn die *LD_PRELOAD*-Umgebungsvariable immer auf den Dateinamen und -pfad unserer Bibliothek zeigt, wird diese vor allen anderen gestarteten Programmen geladen.

Weitere Beispiele:

- Sehr einfache Unterbrechung von strcmp() (Yong Huang) https://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
- Kevin Pulo—Fun with LD_PRELOAD. Viele Beispiele und Ideen. yurichev.com
- Datei-Funktionen unterbrechen beim Komprimieren/Entkomprimieren on-the-fly (zlibc). ftp://metalab.unc.edu/pub/Linux/libs/compression

6.5 Windows NT

6.5.1 CRT (win32)

Startet ein Programm genau bei der `main()`-Funktion? Nein, tut es nicht!

Würden wir jede ausführbare Datei in IDA oder HIEW öffnen können, würde wir sehen, dass OEP auf einen anderen Code-Block zeigt.

Dieser Code erledigt einige Vorbereitungen bevor die Ausführungskontrolle an unseren Code übergeben wird. Dies ist der sogenannte Startup- oder CRT-Code (C Run-Time).

Die `main()`-Funktion nimmt ein Array mit den Argumenten entgegen, die auf der Kommandozeile übergeben wurde, sowie eins mit den Umgebungsvariablen. Genaugenommen wird eine normale Zeichenkette an das Programm übergeben und der CRT-Code unterteilt diesen anhand der Leerzeichen in seine Bestandteile. Der CRT-Code bereitet auch das Array envp vor, welches die Umgebungsvariablen enthält.

Für GUI[14]-Programme unter Win32 wird `WinMain` anstatt `main()` genutzt, welches eigene Argumente hat:

```
int CALLBACK WinMain(
  _In_  HINSTANCE hInstance,
  _In_  HINSTANCE hPrevInstance,
  _In_  LPSTR lpCmdLine,
  _In_  int nCmdShow
);
```

Der CRT-Code bereitet diese ebenfalls vor.

Die Zahl die von `main()` zurückgegeben wird ist der Exit-Code.

Diese kann in der CRT für die Funktion `ExitProcess()` werden, die diesen Exit-Code als Argument entgegennimmt.

In der Regel hat jeder Compiler seinen eigenen CRT-Code.

Nachfolgend eine typischer CRT-Code für MSVC 2008.

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

       push    14h
       push    offset stru_4092D0
       call    __SEH_prolog4
       mov     eax, 5A4Dh
       cmp     ds:400000h, ax
       jnz     short loc_401096
       mov     eax, ds:40003Ch
```

[14] Graphical User Interface

```
       cmp     dword ptr [eax+400000h], 4550h
       jnz     short loc_401096
       mov     ecx, 10Bh
       cmp     [eax+400018h], cx
       jnz     short loc_401096
       cmp     dword ptr [eax+400074h], 0Eh
       jbe     short loc_401096
       xor     ecx, ecx
       cmp     [eax+4000E8h], ecx
       setnz   cl
       mov     [ebp+var_1C], ecx
       jmp     short loc_40109A


loc_401096: ; CODE XREF: ___tmainCRTStartup+18
            ; ___tmainCRTStartup+29 ...
       and     [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
       push    1
       call    __heap_init
       pop     ecx
       test    eax, eax
       jnz     short loc_4010AE
       push    1Ch
       call    _fast_error_exit
       pop     ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
       call    __mtinit
       test    eax, eax
       jnz     short loc_4010BF
       push    10h
       call    _fast_error_exit
       pop     ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
       call    sub_401F2B
       and     [ebp+ms_exc.disabled], 0
       call    __ioinit
       test    eax, eax
       jge     short loc_4010D9
       push    1Bh
       call    __amsg_exit
       pop     ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
       call    ds:GetCommandLineA
       mov     dword_40B7F8, eax
       call    ___crtGetEnvironmentStringsA
       mov     dword_40AC60, eax
       call    __setargv
       test    eax, eax
```

```
        jge     short loc_4010FF
        push    8
        call    __amsg_exit
        pop     ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
        call    __setenvp
        test    eax, eax
        jge     short loc_401110
        push    9
        call    __amsg_exit
        pop     ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
        push    1
        call    __cinit
        pop     ecx
        test    eax, eax
        jz      short loc_401123
        push    eax
        call    __amsg_exit
        pop     ecx

loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
        mov     eax, envp
        mov     dword_40AC80, eax
        push    eax             ; envp
        push    argv            ; argv
        push    argc            ; argc
        call    _main
        add     esp, 0Ch
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN28
        push    eax             ; uExitCode
        call    $LN32

$LN28:      ; CODE XREF: ___tmainCRTStartup+105
        call    __cexit
        jmp     short loc_401186


$LN27:      ; DATA XREF: .rdata:stru_4092D0
        mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
        mov     ecx, [eax]
        mov     ecx, [ecx]
        mov     [ebp+var_24], ecx
        push    eax
        push    ecx
        call    __XcptFilter
        pop     ecx
        pop     ecx

```

```
121 $LN24:
122         retn
123 
124 
125 $LN14:      ; DATA XREF: .rdata:stru_4092D0
126         mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function
    401044
127         mov     eax, [ebp+var_24]
128         mov     [ebp+var_20], eax
129         cmp     [ebp+var_1C], 0
130         jnz     short $LN29
131         push    eax                 ; int
132         call    __exit
133 
134 
135 $LN29:      ; CODE XREF: ___tmainCRTStartup+135
136         call    __c_exit
137 
138 loc_401186: ; CODE XREF: ___tmainCRTStartup+112
139         mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140         mov     eax, [ebp+var_20]
141         call    __SEH_epilog4
142         retn
```

Wir sehen hier die Aufrufe zu GetCommandLineA() (Zeile 62), anschließend zu setargv() (Zeile 66) und setenvp() (Zeile 74), was offensichtlich die globalen Variablen argc, argv und envp initialisiert.

Zum Schluss wird main() mit diesen Argumenten aufgerufen (Zeile 97).

Es sind ebenfalls Aufrufe zu Funktionen mit selbsterklärenden Namen zu finden, wie heap_init() (Zeile 35) und ioinit() (Zeile 54).

Der heap wird jedoch vom CRT initialisiert. Wenn man versucht malloc() in einem Programm ohne CRT zu nutzen, wird dieses mit dem folgenden Fehler abstürzen:

```
runtime error R6030
- CRT not initialized
```

Die Initialisierung von globalen Objekten in C++ passiert ebenfalls in der CRT vor der Ausführung von main(): **??** on page ??.

Ist es möglich die CRT loszuwerden? Ja, wenn man genau weiß, was man tut.

Der Linker von MSVC hat die /ENTRY-Option um den Einsprungpunkt festzulegen.

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Kompilieren wir dies in MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

Wir bekommen eine lauffähige .exe-Datei mit der Größe 2560 Byte mit einem PE-Header, Anweisungen die MessageBox aufrufen, zwei Zeichenketten im Datensegment, die aus user32.dll importierte Funktion MessageBox und sonst nichts.

Dies funktioniert, jedoch kann nicht WinMain mit den 4 Argumenten anstatt main() genutzt werden.

Um genau zu sein, wäre dies zwar möglich, allerdings wurden die Argumente nicht vorbereitet um sie zu nutzen.

Es ist übrigens auch möglich die .exe-Datei kleiner machen indem die PE-Sektion an weniger als den standardmäßigen 4096 Byte auszurichten.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Der Linker gibt aus:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Die Ausgabe ist eine .exe-Datei mit 720 Byte. Sie kann unter Windows 7 x86, jedoch nicht x64 ausgeführt werden (beim Versuch wird eine Fehlermeldung erscheinen).

Mit mehr Aufwand ist es möglich die Datei noch weiter zu verkleinern, aber wie zu sehen ist, bekommt man schnell Kompatibilitätsprobleme.

6.5.2 Win32 PE

PE ist ein Dateiformat für ausführbare Dateien unter Windows. Der Unterschied zwischen .exe, .dll und .sys ist, dass .exe und .sys in der Regel nur Imports und keine Exports haben.

Eine DLL[15] hat wie jede andere PE-Datei einen Eintrittspunkt (OEP) (die Funktion DllMain() befindet sich hier), allerdings macht diese Funktion in der Regel nichts. .sys ist normalerweise ein Gerätetreiber. Wie auch bei Treibern, erwartet Windows eine Prüfsumme in der PE-Datei, die korrekt sein muss[16].

Ab Windows Vista, muss eine Treiberdatei auch mit einer digitalen Signatur versehen sein. Anderseits wird das Laden des Treibers fehlschlagen.

Jede PE-Datei beginnt mit einem kleinen DOS-Programm, welches eine Nachricht in der Art wie folgt ausgibt: „Dieses Programm kann nicht im MS-DOS-Modus gestartet werden.“—wenn versucht wird das Programm unter DOS oder Windows 3.1 zu starten (BSe die das PE-Format nicht kennen).

Terminologie

- Modul: eine separate Datei, .exe oder .dll

[15] Dynamic-Link Library
[16] Hiew(7.1 on page 655) kann diese berechnen

- Prozess: ein Programm das in den Speicher geladen wurde und gerade ausgeführt wird. Besteht meist aus einer .exe-Datei und einer Reihe von .dll-Dateien.
- Prozessspeicher: der Speicher, mit dem der Prozess arbeitet. Jeder Prozess hat seinen eigenen Bereich. Hier befinden sich in der Regel Module, der Stack, Heap(s) usw.
- VA[17]: Eine Adresse welche zur Laufzeit in einem Programm verwendet wird.
- Basisadresse (eines Moduls): die Adresse im Prozessspeicher an der das Modul geladen wird. Der BS-Lader kann diese ändern wenn die Basisadresse bereits von einem vorher geladenen Modul verwendet wird.
- RVA[18]: die VA-Adresse minus der Basisadresse.

  Viele Adressen in PE-Dateien-Tabellen nutzen RVA-Adressen.
- IAT[19]: Ein Array von Adressen importierter Module[20].
  Manchmal zeigt das `IMAGE_DIRECTORY_ENTRY_IAT`-Daten-Verzeichnis auf IAT. Erwähnenswert ist es dass IDA (v6.1) eine Pseudosektion namens `.idata` für IAT allozieren kann, selbst wenn IAT Teil einer anderen Sektion ist.
- INT[21]: Ein Array von Namen zu importierender Symbole[22].

Basisadresse

Das Problem ist, dass mehrere Modulprogrammierer DLL-Dateien für die Nutzung für andere vorbereiten können, es jedoch nicht möglich ist festzulegen welche Adressen für diese Module verwendet werden.

Das ist der Grund, warum in dem Fall wenn zwei für einen Prozess notwendige DLLs dieselbe Basisadresse haben, eine davon an die Basisadresse geladen wird und die andere an eine andere freie Stelle im Prozessspeicher. Jede virtuelle Adresse der zweiten DLL wird korrigiert.

Mit MSVC generiert der Linker oft .exe-Dateien mit der Basisadresse `0x400000`[23], und mit der Code-Sektion die bei `0x401000` beginnt. Dies bedeutet, dass die RVA des Beginns der Code-Sektion `0x1000` ist.

DLLs werden vom MSVC-Linker oft mit der Basisadresse `0x10000000` erzeugt[24].

Es gibt einen weiteren Grund warum Module an verschiedenen Basisadressen, in diesem Fall an zufälligen Adressen, geladen werden: ASLR[25].

Ein Shellcode der auf einem kompromittierten System ausgeführt werden soll, muss Systemfunktionen aufrufen und dementsprechend deren Adressen kennen.

---

[17] Virtual Address
[18] Relative Virtual Address
[19] Import Address Table
[20] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]
[21] Import Name Table
[22] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]
[23] der Ursprung dieser Adress-Auswahl ist hier: MSDN
[24] Dies kann mit der Linker-Option /BASE geändert werden
[25] Address Space Layout Randomization

In älteren BS (in der Windows NT-Reihe: bevor Windows Vista) wurden System-DLL (wie kernel32.dll, user32.dll) immer an bekannte Adressen geladen und mit dem Wissen, dass deren Versionen selten wechseln, waren die Adressen der Funktionen fest und konnten direkt aufgerufen werden.

Um dies zu verhindern läd ASLR das Programm und alle benötigten Module jedes Mal an zufällige Basisadressen.

ASLR-Unterstützung ist in der PE-Datei mit dem Flag `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` markiert [siehe Mark Russinovich, *Microsoft Windows Internals*].

Subsystem

Es existiert auch ein *Subsystem*-Feld, dass normalerweise wie folgt ist:

- native[26] (.sys-Treiber),
- console (Konsolenanwendung) oder
- GUI (Anwendung mit grafischer Oberfläche).

Betriebssystem-Version

Eine PE-Datei spezifiziert auch die minimale Windows-Version die sie benötigt um ladbar zu sein.

Die Tabelle mit Versionsnummern in der PE-Datei und die entsprechenden Windows-Codenamen ist hier[27].

Beispielsweise kompiliert MSVC 2005 .exe-Dateien für Windows NT4 (Version 4.00), MSVC 2008 jedoch nicht (die erzeugten Dateien haben die Version 5.00, es ist mindestens Windows 2000 notwendig um sie ausführen zu können).

MSVC 2012 erzeugt standardmäßig .exe-Dateien mit der Version 6.00, die mindestens Windows Vista benötigen. Mit Änderungen an den Compiler-Option[28] ist es jedoch möglich eine Kompilierung für Windows XP zu erzwingen.

Sektionen

Abschnitte in Sektionen sind in allen Formaten für ausführbare Dateien vorhanden.

Dies wurde entwickelt um Code von Daten und Daten von Konstanten zu trennen.

- Entweder das *IMAGE_SCN_CNT_CODE*- oder *IMAGE_SCN_MEM_EXECUTE*-Flag ist in der Code-Sektion gesetzt. Dies kennzeichnet ausführbaren Code.
- In der Daten-Sektion sind die Flags *IMAGE_SCN_CNT_INITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* und *IMAGE_SCN_MEM_WRITE* gesetzt.

[26] bedeutet, dass das Modul eine eigene API statt Win32 nutzt
[27] Wikipédia
[28] MSDN

- In einer leeren Sektion mit uninitialisierten Daten sind die Flags *IMAGE_SCN_CNT_UNINITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* und *IMAGE_SCN_MEM_WRITE* gesetzt.
- In der Sektion mit konstanten Daten (die schreibgeschützt ist), sind die Flags *IMAGE_SCN_CNT_INITIALIZED_DATA* und *IMAGE_SCN_MEM_READ* gesetzt, nicht jedoch *IMAGE_SCN_MEM_WRITE*. Ein Prozess wird abstürzen, wenn er versucht hier schreibend zuzugreifen.

Jede Sektion in einer PE-Datei kann einen Namen haben, auch wenn dieser nicht unbedingt wichtig ist. Oft (aber nicht immer) ist die Code-Sektion mit `.text` benannt, die Datensektion mit `.data`, und die Sektion mit konstanten Daten `.rdata` für *(readable data)*.

Andere verbreitete Sektionsnamen sind:

- `.idata`: Import-Sektion IDA kann eine Pseudosektion mit dem Namen 6.5.2 on page 612 erzeugen.
- `.edata`: Export-Sektion (selten)
- `.pdata`: Sektion, die alle Informationen über Ausnahmen in Windows NT für MIPS, IA64[29] und x64 enthält: 6.5.3 on page 647
- `.reloc`: Reloc-Sektion
- `.bss`: uninitialisierte Daten (BSS)
- `.tls`: Thread-lokaler Speicher (TLS)
- `.rsrc`: Ressourcen
- `.CRT`: kann in Binärdateien vorhanden sein die mit alten MSVC-Compilern erzeugt wurden

Pack- und Verschlüsselungsprogramme für PE-Dateien verändern häufig die Sektionsnamen oder ersetzen sie durch eigene Namen.

MSVC ermöglicht es Daten in beliebigen benannte Sektion zu deklarieren[30].

Einige Compiler und Linker können eine Sektion mit Debug-Symbolen und anderen Debug-Informationen hinzufügen (MinGW zum Beispiel). Nichtsdestotrotz ist dies in der aktuellen Version von MSVC möglich. Hier gibt es separate PDB-Dateien für diesen Zweck.

Nachfolgend der Aufbau der PE-Sektion in dieser Datei:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
```

[29] Intel Architecture 64 (Itanium)
[30] MSDN

```
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[31]

Ein Wort zur Terminologie: *PointerToRawData* wird in Hiew „Offset“ und *VirtualAddress* „RVA“ genannt.

Datensektion

Die Datensektion in der Datei kann kleiner sein als im Arbeitsspeicher. Zum Beispiel können einige Variablen initialisiert sein und andere nicht. Compiler und Linker sammeln diese alle in einer einzelnen Sektion, aber der erste Teil davon ist initialisiert und in der Datei enthalten, während ein anderer in der Datei fehlt um diese kleiner zu machen. *VirtualSize* wird gleich groß sein wie die Sektion im Speicher und *SizeOfRawData* wie die Sektion in der Datei.

IDA kann die Grenze der initialisierten und nicht initialisierten Teile wie folgt anzeigen:

```
...

.data:10017FFA                 db    0
.data:10017FFB                 db    0
.data:10017FFC                 db    0
.data:10017FFD                 db    0
.data:10017FFE                 db    0
.data:10017FFF                 db    0
.data:10018000                 db    ? ;
.data:10018001                 db    ? ;
.data:10018002                 db    ? ;
.data:10018003                 db    ? ;
.data:10018004                 db    ? ;
.data:10018005                 db    ? ;

...
```

Relocations (relocs)

AKA FIXUPs (zumindest in Hiew). Diese sind ebenfalls in den meisten Formaten für ausführbare Dateien vorhanden[32]. Ausnahmen sind gemeinsam genutzte (dynamische) Bibliotheken, die PIC enthalten

Wofür dienen die relocs?

[31] MSDN
[32] Sogar in .exe-Dateien für MS-DOS.

Offensichtlich können Module an verschiedene Basisadressen geladen werden. Wie wird jedoch zum Beispiel mit globalen Variablen umgegangen? Auf diese muss anhand der Adresse zugegriffen werden. Eine Möglichkeit dazu ist positionsabhängiger Code (6.4.1 on page 600), was aber nicht immer komfortabel ist

Aus diesem Grund existieren Relocation-Tabellen. Hier sind die Adressen die korrigiert werden müssen aufgelistet, falls an eine andere Basisadresse geladen wird.

Beispielsweise ist eine globale Variable an Adresse 0x410000. Das folgende Listing zeigt wie auf diese zugegriffen wird:

```
A1 00 00 41 00         mov         eax,[000410000]
```

Die Basisadresse des Moduls ist 0x400000, die RVA der globalen Variablen ist 0x10000.

Falls das Modul an die Basisadresse 0x500000 geladen wird, muss die reale Adresse der globalen Variablen auf 0x510000 geändert werden.

Wie man sieht ist die Adresse der Variablen in der Anweisung MOV, nach dem 0xA1 kodiert.

Aus diesem Grund wird die Adresse der vier Byte nach 0xA1 in die Relocation-Tabelle geschrieben.

Wenn das Modul an einer anderen Basisadresse geladen wird, listet der BS-Lader alle Adressen in der Tabelle auf, findet jedes 32-Bit-Word auf das die Adresse zeigt, subtrahiert die Basisadresse davon (das ergibt hier die RVA) und addiert die neue Basisadresse hinzu.

Wird das Modul an der originalen Basisadresse geladen passiert nichts.

Alle globalen Variablen können auf diese Weise behandelt werden.

Relocs können verschiedene Typen haben. In Windows für x86-Prozessoren ist dieser üblicherweise *IMAGE_REL_BASED_HIGHLOW*.

Übrigens sind Relocs in Hiew abgedunkelt, wie hier zu sehen: Abb.1.21.

OllyDbg unterstreicht die Orte im Speicher auf die Relocs angewendet wurden, beispielsweise: Abb.1.52.

Exports und Imports

Wie bereits bekannt ist, muss jedes ausführbare Programm in irgendeiner Weise die Dienste des BS oder anderer DLL-Bibliotheken nutzen.

Die Funktionen eines Moduls (in der Regel eine DLL) muss irgendwie mit den Aufrufen in anderen Modulen (.exe-Dateien oder eine andere DLL) verbunden werden.

Aus diesem Grund hat jede DLL eine „Export“-Tabelle die aus Funktionen und deren Adressen in einem Modul besteht.

Außerdem hat jede .exe-Datei oder DLL „Imports“, eine Tabelle von Funktionen und Liste von DLL-Dateinamen, die sie für die Ausführung benötigt.

Nach dem laden der Haupt-exe-Datei verarbeitet der BS-Lader die Import-Tabelle: er lädt die zusätzlichen DLL-Dateien, findet die Funktionsnamen in den DLL-Exports und schreibt deren Adressen in die IAT der .exe-Datei.

Wie man sieht, muss der Lader während seiner Aufgabe viele Funktionsnamen vergleichen. Vergleiche von Zeichenketten sind jedoch nicht sehr performant, so dass es Unterstützung für „ordinals“ oder „hints“ gibt, welche aus Funktionsnamen in der Tabelle bestehen statt deren Namen.

Auf diese Weise können sie beim Laden einer DLL schneller gefunden werden.

Die „ordinals“ also Zahlworte, sind in der Export-Tabelle immer vorhanden.

Auf diese Weise laden Programme welche die MFC[33]-Bibliothek nutzen die mfc*.dll und es existieren keine MFC-Namen wie in INT.

Wenn solche Programme in IDA geladen werden, wird nach dem Pfad zu den mfc*.dll-Dateien gefragt um die Funktionsnamen herausfinden zu können.

Wenn der Pfad zu diesen DLLs in IDA nicht angegeben wird, erscheint *mfc80_123* statt der Funktionsnamen.

Import-Sektion

Häufig wird eine separate Sektion mit dem Namen `.idata` für die Import-Tabelle und alle dafür relevanten Dinge angelegt. Dies ist aber keine strikte Regel.

Imports sind manchmal etwas verwirrend wegen der uneinheitlichen Terminologie. Versuchen wir alle Informationen an einer Stelle zu sammeln.

[33]Microsoft Foundation Classes

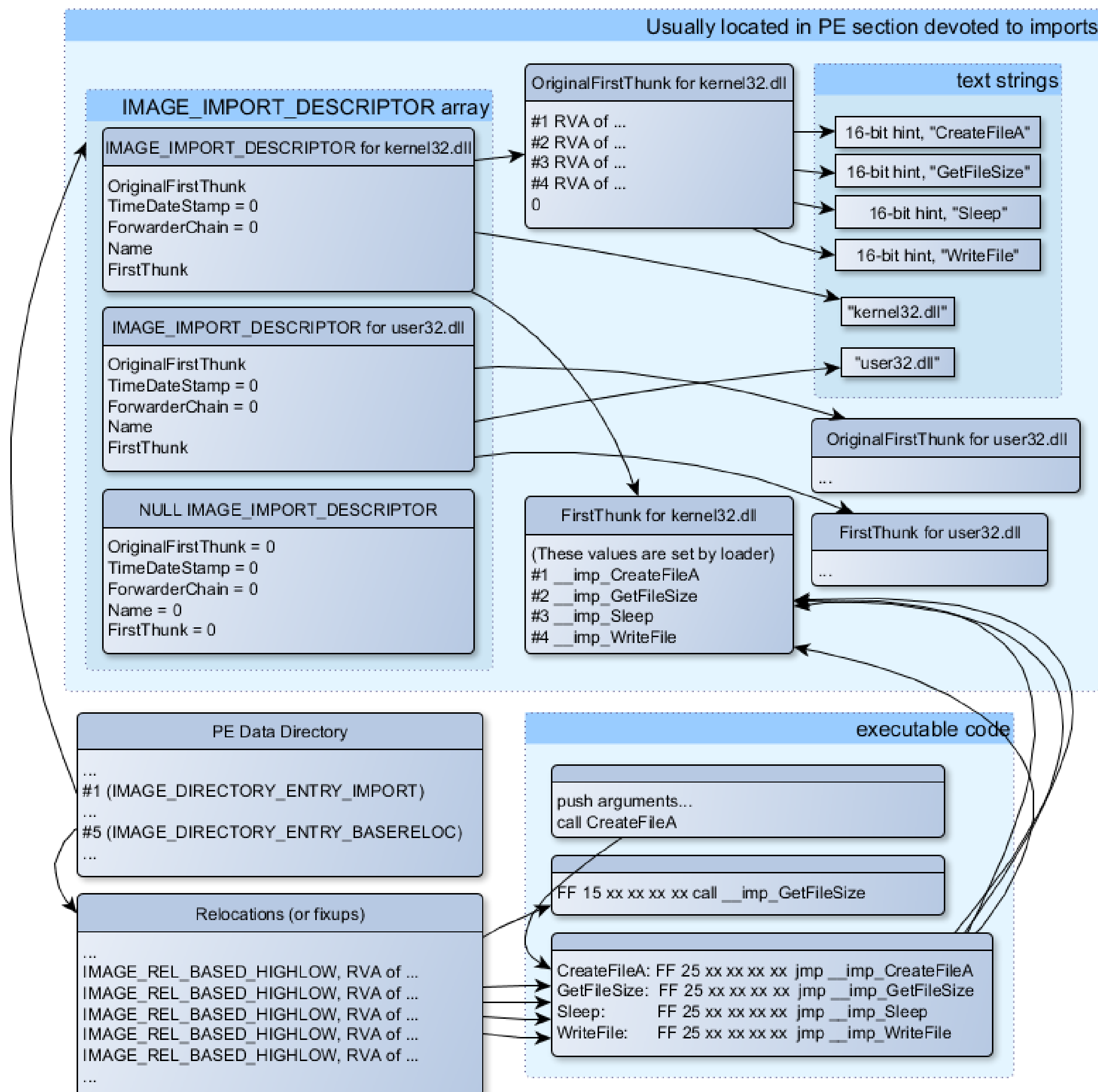


Abbildung 6.1: Ein Schema dass alle PE-Datei-Strukturen im Zusammenhang mit Imports vereint

Die Hauptstruktur ist das Array *IMAGE_IMPORT_DESCRIPTOR*. Jedes Element für jede DLL wird importiert.

Jedes Element hat die RVA-Adresse der Zeichenkette (DLL-Name) (*Name*).

OriginalFirstThunk ist die RVA-Adresse der INT-Tabelle. Diese ist ein Array von RVA-Adressen, jede davon zeigt auf eine Zeichenkette mit einem Funktionsnamen. Jede Zeichenkette wird eine 16-Bit-Integerzahl voran gestellt („hint")—„ordinal" der Funktion).

Während des Ladens und falls es möglich ist die Funktion anhand der Zahl zu finden, wird der Vergleich der Zeichenketten nicht auftauchen. Das Array ist mit Null terminiert.

Es gibt auch einen Zeiger zur IAT-Tabelle mit dem Namen *FirstThunk*. Dies entspricht der RVA-Adresse der Stelle an der der Lader die Adressen der aufgelösten Funktionen schreibt.

Die Punkte an denen der Lader die Adressen schreibt werden in IDA mit *__imp_CreateFileA* und so weiter gekennzeichnet.

Es gibt zumindest zwei Arten die vom Lader geschriebenen Adressen zu nutzen.

- Der Code enthält eine Anweisung wie *call __imp_CreateFileA*. Da das Feld mit der Adresse der importierten Funktion in gewisser Weise eine globale Variable ist, wird die Adresse der *call*-Anweisung (plus 1 oder 2) zur Relocation-Tabelle hinzugefügt. Dies gilt für den Fall falls das Modul an eine andere Basisadresse geladen wird.

  Aber offensichtlich kann dies die Relocation-Tabelle erheblich vergrößern, da möglicherweise in dem Modul viele Aufrufe von importierten Funktionen enthalten sind.

  Des weiteren verlangsamen die großen Tabellen das Laden der Module.

- Für jede importierte Funktion wird nur ein Sprung mittels der JMP-Anweisung und einem Reloc darauf alloziert. Solche Punkte werden auch „thunk“s genannt.

  Alle Aufrufe zu den importierten Funktionen sind lediglich CALL-Anweisungen auf die entsprechenden „thunks“. In diesem Fall sind keine zusätzlichen Relocs notwendig, da diese CALLs eine relative Adresse haben und nicht korrigiert werden müssen.

Diese beiden Methoden können miteinander kombiniert werden.

Der Linker kann mehrere einzelne „thunk“s erstellen, falls zu viele Aufrufe der Funktion vorliegen. Dies ist jedoch nicht das Standardverhalten.

Übrigens muss das Array der Funktionsadressen auf das FirstThunk zeigt nicht unbedingt in der IAT-Sektion sein. Beispielsweise hat der Autor dieser Zeilen einmal das PE_add_import[34]-Tool geschrieben um Imports zu einer existierenden .exe-Datei hinzufügen zu können.

In einer früheren Version des Tools hat dieses an die Stelle der Funktion an der ein Aufruf zu einer anderen DLL geschrieben werden sollte, folgenden Code erzeugt:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk zeigt auf die erste Anweisung. Mit anderen Worten, wenn yourdll.dll geladen wird, schreibt der Lader die Adresse der Funktion *function* direkt in den Code.

Erwähnenswert ist, dass die Code-Sektion normalerweise schreibgeschützt ist. Daher fügt das Tool das *IMAGE_SCN_MEM_WRITE*-Flag hinzu, da andernfalls das Programm beim Laden mit dem Fehlercode 5 (Zugriff verweigert.) abstürzen würde.

[34] yurichev.com

Man mag sich fragen: was ist wenn ein Programm mit ein paar DLLs kommt, die sich nicht ändern (inklusive der Adressen aller DLL-Funktionen). Ist es möglich den Lade-Vorgang zu beschleunigen?

Dies ist in der Tat möglich, wenn die Adressen der Funktionen bereits im Voraus in das FirstThunk-Array geschrieben werden.
Das *Timestamp*-Feld ist in der *IMAGE_IMPORT_DESCRIPTOR*-Struktur vorhanden.

Wenn der Wert dort verfügbar ist, vergleicht der Lader diesen Wert mit dem Zeitstempel der DLL-Datei.

Wenn der Wert gleich ist, macht der Lader nichts und der Vorgang kann schneller sein. Dies wird „old-style binding“[35] genannt.

Das Tool BIND.EXE ist für diesen Vorgang gedacht.

Um das Laden eigener Programme zu beschleunigen empfiehlt Matt Pietrek in Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)][36]das Binden kurz nach der Installation des Programms auf dem Rechner des Endanwenders durchzuführen.

Komprimierungs- und Verschlüsselungsprogramme für PE-Dateien komprimieren bzw. Verschlüsseln auch die Import-Tabellen.

In diesem Fall wird der Windows-Lader natürlich nicht alle notwendigen DLL-Dateien lasen.

Der Komprimierer / Verschlüsseler macht dies selber mithilfe der Funktionen *LoadLibrary()* und *GetProcAddress()*.

Aus diesem Grund sind die beiden Funktionen oft im IAT von gepackten Dateien.

In den Standard-DLLs der Windows-Installation ist IAT häufig zu Beginn einer PE-Datei zu finden. Vermutlich geschieht dies aus Optimierungsgründen.

Während die .exe-Datei geladen wird, befindet sich diese nicht als Ganzes im Speicher (man denke an riesige Installationsprogramme welche verdächtig schnell geladen werden), sondern ist „gemapped“ und wird in Teilen geladen, wenn auf diese zugegriffen wird.

Vielleicht waren die Microsoft-Entwickler der Meinung, dass dies schneller ist.

Ressourcen

Ressourcen in einer PE-Datei sind lediglich Sammlungen von Icons, Bildern, Zeichenketten und Dialog-Beschreibungen.

Möglicherweise wurden Sie vom Hauptcode getrennt um mehrere Sprachen unterstützen zu können und es einfacher ist einen Text oder ein Bild in der Sprache auszuwählen, die zur Zeit im BS eingestellt ist.

Ein Seiteneffekt ist, dass diese einfach editiert und in der ausführbaren Datei zurück gespeichert werden können. Mit speziellen Editoren wie beispielsweise (6.5.2 on the next page) ist dies auch ohne spezielles Wissen möglich.

[35] MSDN. Dort auch „new-style binding“.
[36] Auch verfügbar als http://msdn.microsoft.com/en-us/magazine/bb985992.aspx

.NET

.NET-Progamme werden nicht in Maschinencode übersetzt sondern in einen speziellen Bytecode. Streng genommen befindet sich anstatt des gewöhnlichen x86-Code der Bytecode in der .exe-Datei. Der Einsprungpunkt (OEP) jedoch zeigt auf ein kleines Fragment x86-Code:

```
jmp         mscoree.dll!_CorExeMain
```

Der .NET-Lader befindet sich in mscoree.dll, welche die PE-Datei verarbeitet.

Dies war in allen Windows-Versionen vor Windows XP der Fall. Seit Windows XP ist der BS-Lader in der Lage .NET-Dateien zu erkennen und diese ohne eine JMP-Anweisung auszuführen[37].

TLS

Diese Sektion beinhaltet initialisierte Daten für TLS(6.2 on page 591) (falls notwendig). Wird ein neuer Thread gestartet, werden dessen TLS-Daten mit den Daten dieser Sektion initialisiert.

Abgesehen davon beinhaltet die Spezifikation für PE-Dateien auch die Möglichkeit der Initialisierung der TLS-Sektion, sogenannte TLS-Callbacks.

Falls diese vorhanden sind, werden sie aufgerufen bevor die Ausführungskontrolle an den Haupteinsprungpunkt (OEP) übergeben wird.

Dies ist sehr verbreitet bei Packern und Verschlüsselungsprogrammen für PE-Dateien.

Tools

- objdump (in Cygwin enthalten) um alle PE-Dateistrukturen auszugeben.
- Hiew(7.1 on page 655) als Editor.
- pefile: Python-Bibliothek für die Verarbeitung von PE-Dateien[38].
- ResHack AKA Resource Hacker: Ressourcen-Editor[39].
- PE_add_import[40]: einfaches Tool um Symbole zur PE Importtabelle hinzuzufügen.
- PE_patcher[41]: einfaches Tool um ausführbare PE-Dateien zu patchen.
- PE_search_str_refs[42]: einfaches Tool zum Suchen von Funktionen in ausführbaren PE-Dateien die bestimmte Zeichenketten nutzen.

[37] MSDN
[38] https://code.google.com/p/pefile/
[39] https://code.google.com/p/pefile/
[40] http://yurichev.com/PE_add_imports.html
[41] yurichev.com
[42] yurichev.com

weitere Informationen

- Daniel Pistelli: The .NET File Format [43]

6.5.3 Windows SEH

Vergessen wir MSVC

Unter Windows ist der Zweck von SEH[44] die Ausnahmebehandlung. Nichtsdestotrotz ist es sprachunabhängig und nicht in irgendeiner Weise an C++ oder OOP[45] gebunden.

Wir betrachten SEH hier in einer isolierten Form und nicht im Zusammenhang mit C++ oder MSVC-Erweiterungen.

Jeder laufende Prozess hat eine Kette von SEH-Handles, TIB beinhaltet die Adresse des letzten Handlers.

Wenn eine Ausnahme auftritt (Division durch Null, Zugriff auf fehlerhafte Adresse, Benutzer-Ausnahme durch Aufruf `RaiseException()`-Funktion), findet das BS den letzten Handler in der TIB und ruft ihn auf. Dabei werden alle Informationen über den Zustand der CPU (Register-Werte usw.) im Moment der Ausnahme übergeben.

Wenn der Ausnahme-Handler eine bekannte Ausnahme sieht wird sie von ihm behandelt.

Ist dies nicht der Fall, wird das BS darüber informiert, dass keine Ausnahmebehandlung stattfand und das BS ruft den nächsten Handler in der Kette, bis ein Handler gefunden wird, der die Ausnahme behandeln kann.

Am Ende der Kette ist ein Standard-Handler, der den wohlbekannten Dialog anzeigt, welcher den Benutzer über den Prozessabsturz informiert. Zusätzlich werden einige technische Informationen wie der CPU-Status beim Zeitpunkt des Absturzes und die Möglichkeit zum Senden der Infos an Microsoft-Entwickler angezeigt.

[43] http://www.codeproject.com/Articles/12585/The-NET-File-Format
[44] Structured Exception Handling
[45] Objektorientierte Programmierung

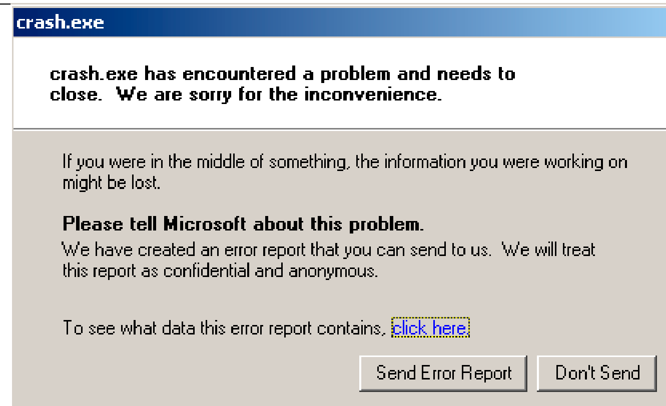


Abbildung 6.2: Windows XP

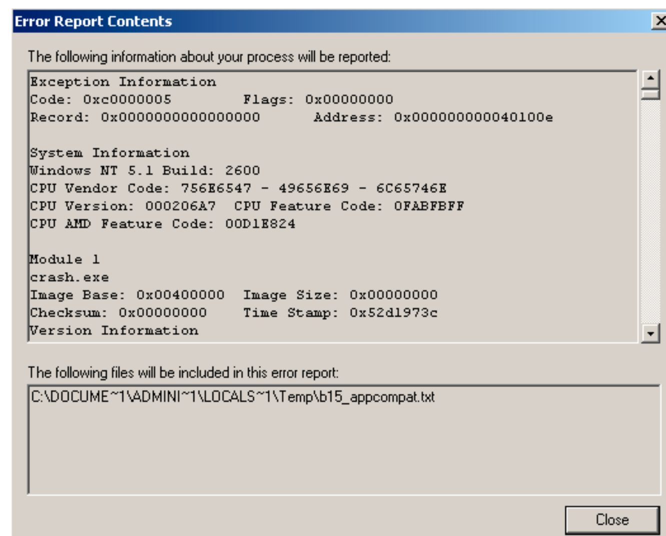


Abbildung 6.3: Windows XP

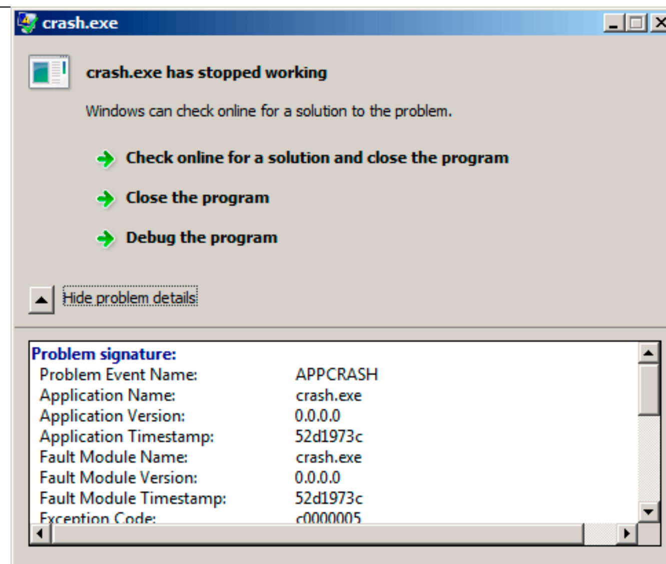


Abbildung 6.4: Windows 7

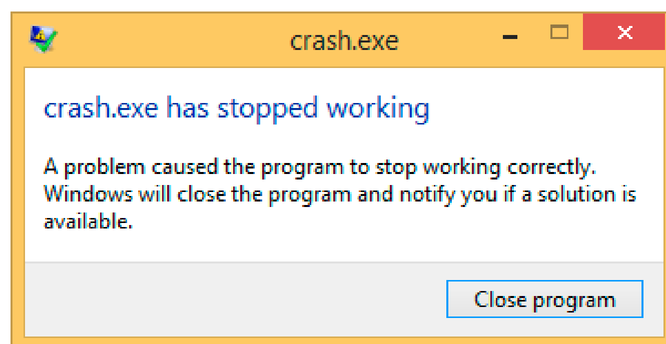


Abbildung 6.5: Windows 8.1

Früher wurde dieser Handler Dr. Watson genannt.

Einige Entwickler erstellen ihren eigenen Handler, der Informationen über den Absturz des Programms zu ihnen selbst schickt. Dieser wird mit der Funktion `SetUnhandledExceptionFil` registriert und aufgerufen, wenn das BS keine andere Möglichkeit hat die Ausnahme zu behandeln. Ein Beispiel ist Oracle RDBMS die eine riesige Menge an möglichen Informationen über die CPU und den Zustand des Speichers sammelt.

Nachfolgend wird ein eigener, einfacher Ausnahme-Handler erstellt. Dieses Beispiel basiert auf dem Beispiel von [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][46]und muss mit der SAFESEH-Option kompiliert werden: `cl seh1.cpp /link /safeseh:no`. Mehr über SAFESEH befindet sich hier: MSDN.

[46]Auch verfügbar als `http://www.microsoft.com/msj/0197/Exception/Exception.aspx`

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
{
        unsigned i;

        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->↙
    ↳ ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->↙
    ↳ ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord↙
    ↳ ->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION↙
    ↳ )
        {
                printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n");
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer to our
    handler

        // install exception handler
        __asm
        {                                // make EXCEPTION_REGISTRATION
```

```
    record:
                push    handler         // address of handler function
                push    FS:[0]          // address of previous handler
                mov     FS:[0],ESP      // add new EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                               // remove our EXECEPTION_REGISTRATION
    record
                mov     eax,[ESP]       // get pointer to previous record
                mov     FS:[0], EAX     // install previous record
                add     esp, 8          // clean our EXECEPTION_REGISTRATION
    off stack
        }

        return 0;
}
```

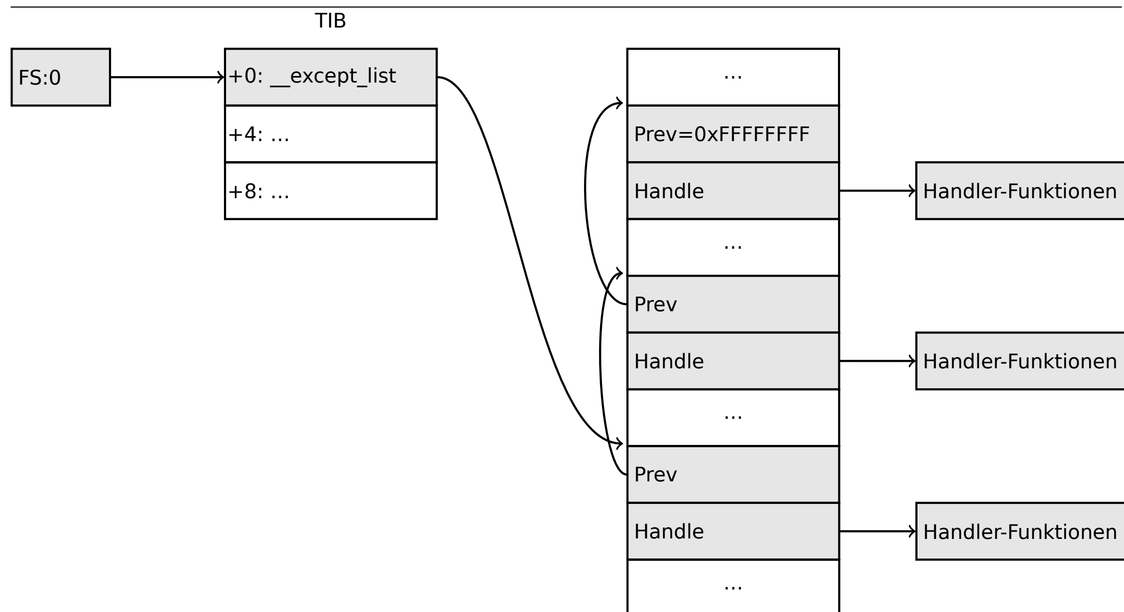
Das FS:Segment-Register zeigt unter Win32 auf die TIB.

Das erste Element der TIB ist ein Zeiger auf den letzten Handler der Kette. Wir sichern den Stack und speichern hier die Adresse unseres Handlers. Die Struktur heißt _EXCEPTION_REGISTRATION. Dabei handelt es sich um eine einfach-verkettete Liste, deren Elemente direkt auf dem Stack gesichert werden.

Listing 6.22: MSVC/VC/crt/src/exsup.inc

```
_EXCEPTION_REGISTRATION struc
     prev    dd      ?
     handler dd      ?
_EXCEPTION_REGISTRATION ends
```

Jedes „handler“-Feld zeigt auf einen Handler und jedes „prev“-Feld zeigt auf den vorherigen Eintrag auf dem Stack. Der letzte Eintrag hat 0xFFFFFFFF (-1) im „prev“-Feld.



Nachdem unser Handler installiert wurde, wird `RaiseException()` [47] aufgerufen. Dies ist eine Benutzer-Ausnahme. Der Handler überprüft diesen Code. Ist der Code `0xE1223344`, wird `ExceptionContinueExecution` zurückgegeben, was bedeutet, dass der CPU-Zustand korrigiert wurde (in der Regel in den EIP-/ESP-Registern) und anschließend das BS die Ausführung fortsetzen kann. Wenn der Code leicht verändert wird, so gibt der Handler `ExceptionContinueSearch` zurück, und das BS wird andere Handler aufrufen. Es ist unwahrscheinlich, dass ein Handle gefunden werden kann, weil keine Informationen (oder Quellcode) darüber vorliegt. Der Standard-Windows-Dialog wird mit einem Hinweis auf einen Prozess-Absturz aufgerufen.

Was ist der Unterschied zwischen einer System-Ausnahme und einer User-Ausnahme? Hier sind die Ausnahmen des Systems:

[47] MSDN

win in WinBase.h definiert	wie in ntstatus.h definiert	Wert
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

Nachfolgend wie der Code aufgebaut ist:

31	29	28	27 … 16	15 … 0
S	U	0	Facility code	Fehler-Code

S ist ein einfacher Status-Code: 11—Fehler; 10—Warnung; 01—Information; 00—Erfolgreich. U—Kennzeichnet ob es sich um User-Code handelt.

Dies erklärt, warum wir oben den Wert 0xE1223344 gewählt haben—E_{16} (1110_2) 0xE (1110b) bedeutet, dass es sich um eine User-Ausnahme handelt; 2) ein Fehler vorliegt.

Genaugenommen funktioniert dieses Beispiel jedoch auch gut ohne diese höherwertigen Bits.

Anschließend versuchen wir einen Wert von der Speicheradresse 0 zu lesen.

Natürlich befindet sich hier nichts unter Win32, womit eine Ausnahme geworfen wird.

Der allererste Handler wird aufgerufen (der von oben) und prüft ob der Code der Konstante
`EXCEPTION_ACCESS_VIOLATION` entspricht.

Der Code der von der Adresse an der Speicherstelle 0 liest, sieht wie folgt aus:

Listing 6.23: MSVC 2010

```
        ...
        xor     eax, eax
        mov     eax, DWORD PTR [eax] ; exception will occur here
```

```
        push    eax
        push    OFFSET msg
        call    _printf
        add     esp, 8
        ...
```

Ist es möglich diese Fehler „on the fly" zu beheben und mit der Programmausführung fortzufahren?

In der Tat ist dies möglich, da unser Ausnahme-Handler den EAX-Wert beheben und das BS diese Anweisung ein weiteres Mal ausführen kann. Das ist was wir tun. `printf()` gibt 1234 aus, weil EAX nach der Ausnahmebehandlung nicht mehr 0 ist sondern die Adresse der globalen Variable `new_value` enthält. Die Ausführung wird fortgesetzt.

Das ist was passiert: die Speicherverwaltung in der CPU signalisiert einen Fehler und die CPU stoppt den Thread, findet die passende Ausnahmebehandlung im Windows-Kernel welche wiederum nacheinander alle Handler in der SEH-Kette aufruft.

Hier wird MSVC 2010 genutzt, aber es gibt natürlich keine Garantie, dass EAX für diesen Zeiger genutzt wird.

Dieser Adresse-Ersatz-Trick dient der Veranschaulichung der internen Vorgänge von SEH. Dennoch ist es schwierig einen realen Einsatz für das Fixen eines Fehlers „on-the-fly" zu finden.

Warum sind die SEH-Einträge direkt auf dem Stack gespeichert und nicht irgendwo anders?

Vermutlich ist der Grund, weil das BS sich dann nicht um das freigeben der Informationen kümmern muss. Diese Einträge werden nach dem Ende der Funktion automatisch gesäubert. Dies entspricht in gewisser Weise alloca(): (1.7.3 on page 47).

Zurück zu MSVC

Offensichtlich benötigten die Microsoft-Entwickler Ausnahmen in C aber nicht in C++ und führten eine nicht-standardisierte C-Erweiterung ein [48]. Diese hat aber keinen Zusammenhang zu C++ PS-Ausnahmen.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

Der „Finally"-Block kann anstelle des Handler-Codes stehen:

```
__try
{
    ...
```

[48] MSDN

```
}
__finally
{
    ...
}
```

Der Filter-Code ist ein Ausdruck, der anzeigt, ob dieser Handler-Code zu der geworfenen Ausnahme passt.

If der Code zu groß ist und nicht in einen Ausdruck passt, kann eine separate Filter-Funktion definiert werden.

Im Windows-Kernel existieren eine Reihe solcher Konstrukte. Nachfolgend einige Beispiel von dort (WRK):

Listing 6.24: WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Listing 6.25: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                    (PAGE_SIZE - PageOffset) :
                    (ReceivedLength - PageOffset) );

} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                         &Status ) ) {
```

Hier ist ein Filter-Code-Beispiel:

Listing 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )
```

```
/*++

Routine Description:

    This routine serves as an exception filter and has the special job of
    extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
                    status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
         (ExceptionPointer->ExceptionRecord->NumberParameters >= 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->ExceptionRecord->↙
    ↳ ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

Intern ist SEH eine Erweiterung der vom BS-unterstützten Ausnahmen, aber die Handler-Funktion ist `_except_handler3` (für SEH3) oder `_except_handler4` (für SEH4).

Der Code dieses Handlers ist MSVC-spezifisch und befindet sich in dessen Bibliotheken oder in der msvcr*.dll. Es ist wichtig zu wissen, dass SEH eine MSVC-spezifische Sache ist.

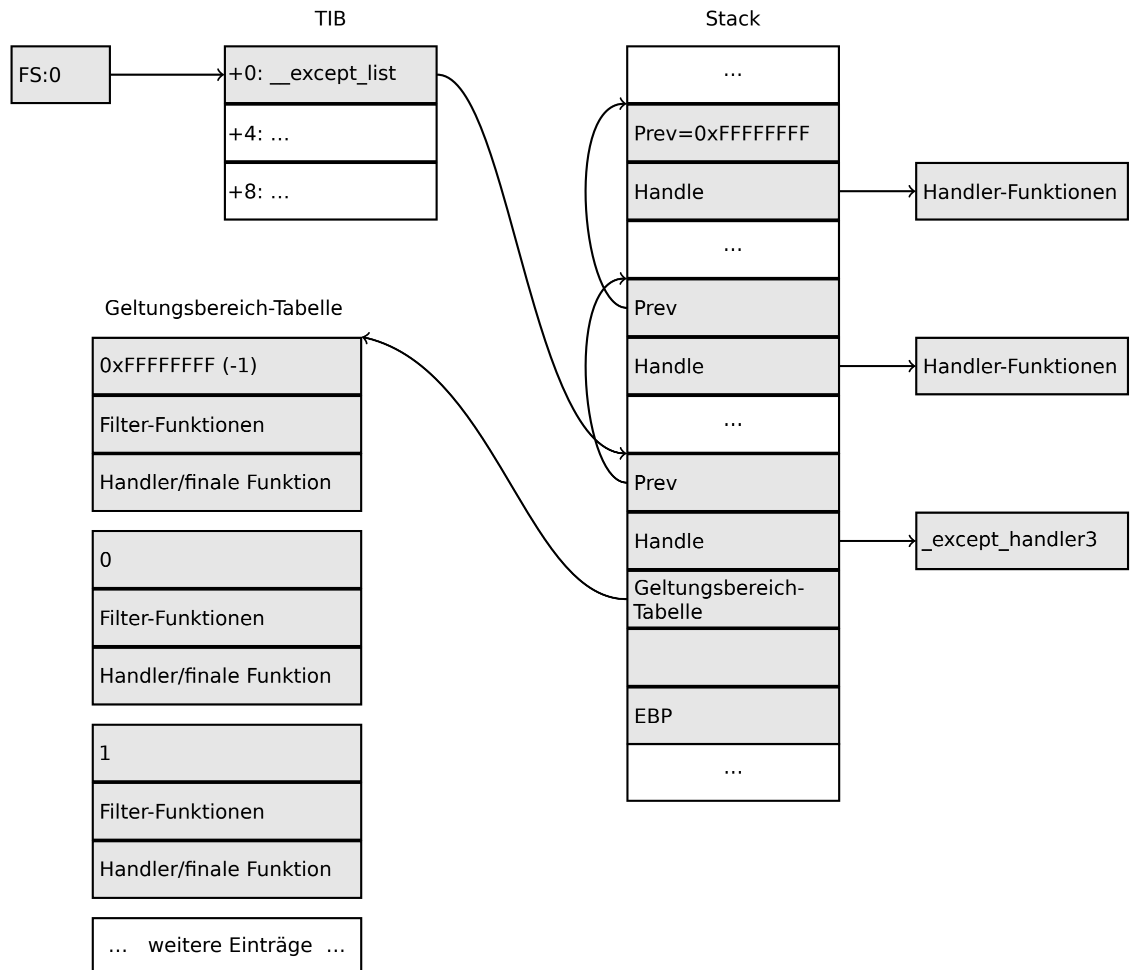
Andere Win32-Compiler bieten möglicherweise etwas völlig anderes an.

SEH3

SEH3 hat `_except_handler3` als Handler-Funktion und erweitert die `_EXCEPTION_REGISTRATION`-Tabelle indem ein Zeiger zur *Scope-Tabelle* und der *previous try level*-Variablen hin-

zugefügt wird. SEH4 erweitert die *Scope-Tabelle* um vier Werte für Schutz vor Speicherüberläufen.

Die *Scope-Tabelle* ist eine Tabelle die aus Zeigern auf Filter und Handler-Code-Blöcken für jede verschachtelte Ebene für *try/except* besteht.



Auch hier ist es wieder sehr wichtig zu verstehen, dass das BS sich lediglich um die *prev/handle*-Felder kümmert und sonst nichts.

Es ist Aufgaben der `_except_handler3`-Funktion die anderen Felder und die *Scope-Tabelle* zu lesen und zu entscheiden, welcher Handler wann aufgerufen werden muss.

Der Quellcode der `_except_handler3`-Funktion ist nicht offen.

Sanos OS, welches einen Win32-Kompatibilitäts-Layer hat, hat die gleiche Funktion implementiert, welche ähnlich ist zu der unter Windows[49]. Eine weitere Implemen-

[49] https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c

tierung existiert in Wine[50] und ReactOS[51].

Wenn der *Filter*-Zeiger NULL ist, ist der *Handler*-Zeiger ein Zeiger auf den *finally*-Code-Block.

Während der Ausführung verändert sich der Wert des *previous try level*, so dass der _except_handler3 Information über den aktuellen Verschachtelungslevel hat, um zu wissen welcher Eintrag der *Scope-Tabelle* zu nutzen ist.

SEH3: one try/except block example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 6.27: MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table:
CONST     SEGMENT
$T74622   DD    0ffffffffH    ; previous try level
          DD    FLAT:$L74617  ; filter
          DD    FLAT:$L74618  ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
```

[50] GitHub
[51] http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html

```
    push   ebp
    mov    ebp, esp
    push   -1                            ; previous try level
    push   OFFSET FLAT:$T74622           ; scope table
    push   OFFSET FLAT:__except_handler3   ; handler
    mov    eax, DWORD PTR fs:__except_list
    push   eax                           ; prev
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -16
; 3 registers to be saved:
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0  ; previous try level
    push   OFFSET FLAT:$SG74605 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET FLAT:$SG74606 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try level
    jmp    SHORT $L74616

    ; filter code:
$L74617:
$L74627:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74619:
$L74626:
    ret    0

    ; handler code:
$L74618:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous try level back
    to -1
$L74616:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
```

```
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Hier ist zu sehen wie der SEH-Frame auf dem Stack aufgebaut ist. Die *Scope-Tabelle* befindet sich im CONST-Segment, diese Felder werden nicht verändert. Eine interessante Sache ist es, wie die *previous try level*-Variable sich geändert hat. Der Wert zu Beginn ist 0xFFFFFFFF (-1). Der Moment, in dem der Body der try-Anweisung betreten wird, ist mit einer Anweisung gekennzeichnet, die 0 in die Variable schreibt. In dem Moment in dem der Body der try-Anweisung geschlossen wird, wird der Wert -1 dorthin zurückgeschrieben. Es sind ebenso die Adressen der Filter- und Handler-Codes zu sehen.

Wir können sehr einfach die Struktur des *try/except*-Konstrukts in der Funktion erkennen.

Da der SEH-Setup-Code im Funktionsprolog von mehreren Funktionen geteilt werden kann, fügt der Compiler manchmal einen Aufruf zur SEH_prolog()-Funktion in den Prolog ein, welcher genau dieses tut.

Der SEH-Aufräumcode ist in der SEH_epilog()-Funktion.

Versuchen wir dieses Beispiel in tracer laufen zu lassen:

```
tracer.exe -l:2.exe --dump-seh
```

Listing 6.28: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ⤦
    ↳ ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!⤦
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0⤦
    ↳ x60) handler=0x401088 (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!⤦
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!⤦
    ↳ mainCRTStartup+0x18d) handler=0x401545 (2.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!⤦
    ↳ __except_handler4)
SEH4 frame. previous trylevel=0
```

```
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb ntdll.dll!↙
    ↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!↙
    ↳ _FinalExceptionHandler@16)
```

Es ist zu erkennen, dass die SEH-Kette aus vier Handlern besteht.

Die ersten zwei sind in unserem Beispiel zu finden. Zwei? Es wurde doch nur einer erstellt?! Das stimmt, jedoch wurde ein weiterer in der CRT-Funktion `_mainCRTStartup()` erstellt und es scheint so, dass hier zumindest FPU-Ausnahmen behandelt. Der Quellcode kann in der MSVC-Installation gefunden werden: `crt/src/winxfltr.c`.

Der dritte ist SEH4 in ntdll.dll und der vierte Handler ist nicht MSVC-spezifisch, befindet sich in der ntdll.ll und hat einen selbsterklärenden Funktionsnamen.

Es ist zu erkennen, dass es drei Arten von Handlern in einer Kette gibt:

Einer ist in keiner Weise in Verbindung zu MVSC (der letzte) und zwei sind MSVC-spezifisch: SEH3 und SEH4.

SEH3: two try/except blocks example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *↙
    ↳ ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
```

```
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
                EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), ↙
    ↘ GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Es existieren jetzt zwei `try`-Blöcke. Die *Scope-Tabelle* hat jetzt zwei Einträge, einen für jeden Block. *Previous try level* verändert sich wenn die Ausführung einen `try`-Block betritt oder verlässt.

Listing 6.29: MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB    'access violation, can''t recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8   ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push   OFFSET FLAT:$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
```

```
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table:
CONST    SEGMENT
$T74644   DD    0ffffffffH   ; previous try level for outer block
          DD    FLAT:$L74634 ; outer block filter
          DD    FLAT:$L74635 ; outer block handler
          DD    00H          ; previous try level for inner block
          DD    FLAT:$L74638 ; inner block filter
          DD    FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT:$T74644
    push   OFFSET FLAT:__except_handler3
    mov    eax, DWORD PTR fs:__except_list
    push   eax
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block entered. set
    previous try level to 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block entered. set
    previous try level to 1
    push   OFFSET FLAT:$SG74617 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867    ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set
    previous try level back to 0
```

```
    jmp     SHORT $L74615

; inner block filter:
$L74638:
$L74650:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74643[ebp], eax
    mov     eax, DWORD PTR $T74643[ebp]
    sub     eax, -1073741819; c0000005H
    neg     eax
    sbb     eax, eax
    inc     eax
$L74640:
$L74648:
    ret     0

; inner block handler:
$L74639:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74621 ; 'access violation, can''t recover'
    call    _printf
    add     esp, 4
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block exited. set
    previous try level back to 0

$L74615:
    mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block exited, set
    previous try level back to -1
    jmp     SHORT $L74633

; outer block filter:
$L74634:
$L74651:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T74642[ebp], eax
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    push    ecx
    mov     edx, DWORD PTR $T74642[ebp]
    push    edx
    call    _filter_user_exceptions
    add     esp, 8
$L74636:
$L74649:
    ret     0

; outer block handler:
$L74635:
    mov     esp, DWORD PTR __$SEHRec$[ebp]
    push    OFFSET FLAT:$SG74623 ; 'user exception caught'
    call    _printf
```

```
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks exited. set
    previous try level back to -1
$L74633:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Wenn ein Breakpoint auf die `printf()`-Funktion gesetzt wird, die vom Handler aufgerufen wird, ist auch sichtbar, wie ein neuer SEH-Handler hinzugefügt wird.

Möglicherweise ist innerhalb des SEH Handling-Prozesses noch eine andere Funktion. Es sind hier in der *Scope-Tabelle* zwei Einträge zu sehen.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

Listing 6.30: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!↙
    ↳ ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0↙
    ↳ xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78↙
    ↳ ) handler=0x401100 (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!↙
    ↳ mainCRTStartup+0x18d) handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!↙
    ↳ __except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!↙
    ↳ ___safe_se_handler_table+0x20) handler=0x771f90eb ntdll.dll!↙
    ↳ _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!↙
    ↳ _FinalExceptionHandler@16)
```

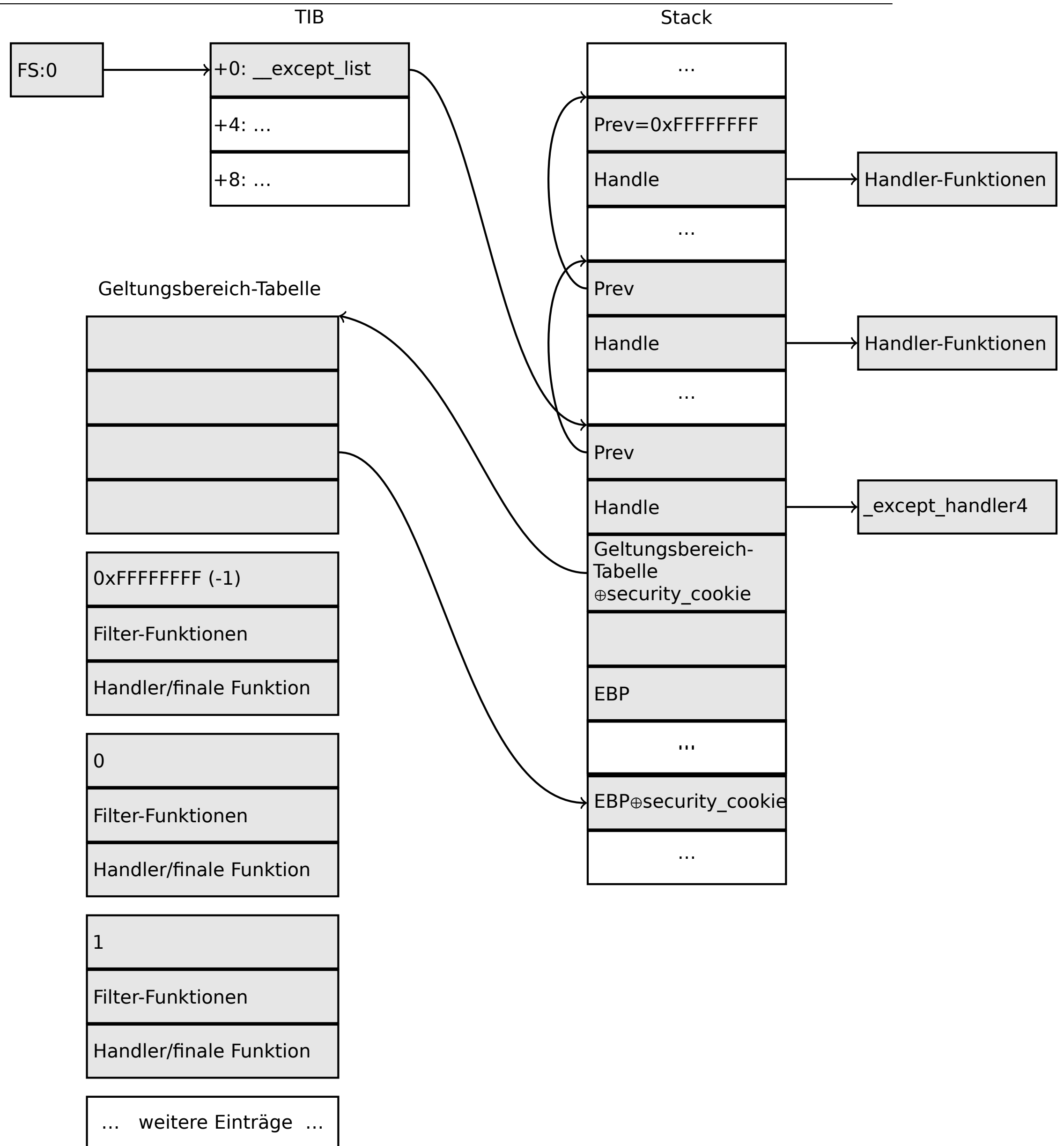
SEH4

Bei einer Pufferüberlauf-Attacke (1.20.2 on page 318), kann die Adresse der *Scope-Tabelle* überschrieben werden. Aus diesem Grund wird seit MSVC 2005 SEH3 auf SEH4 aktualiiert um einen Schutz gegen diese Attacken zu haben. Der Zeiger auf die *Scope-Tabelle* wird jetzt mit einem Security-Cookie xored.

Jedes Element hat einen Offset innerhalb des Stacks mit einem anderen Wert: die Adresse des Stack Frame (EBP) xored mit dem Security-Cookie im Stack.

Dieser Wert wird während der Ausführung der Ausnahmebehandlung ausgelesen und auf Korrektheit überprüft. Das *Security-Cookie* im Stack ist jedes Mal zufällig, so dass ein Angreifer den Wert hoffentlich nicht voraussehen kann.

Der initiale *previous try level* ist -2 in SEH4 anstatt -1.



Hier sind beide Beispiele mit MSVC 2012 und SEH4 kompiliert:

Listing 6.31: MSVC 2012: one try block example

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table:
xdata$x          SEGMENT
__sehtable$_main DD 0fffffffeH   ; GS Cookie Offset
    DD           00H             ; GS Cookie XOR Offset
```

```
    DD          0ffffffccH      ; EH Cookie Offset
    DD          00H             ; EH Cookie XOR Offset
    DD          0fffffffeH      ; previous try level
    DD          FLAT:$LN12@main ; filter
    DD          FLAT:$LN8@main  ; handler
xdata$x         ENDS

$T2 = -36         ; size = 4
_p$ = -32         ; size = 4
tv68 = -28        ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor    eax, ebp
    push   eax                              ; ebp ^ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ;
    pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
```

```
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0

; handler:
$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Listing 6.32: MSVC 2012: two try blocks example

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0fffffffeH         ; GS Cookie Offset
                 DD    00H              ; GS Cookie XOR Offset
                 DD    0ffffffc8H       ; EH Cookie Offset
                 DD    00H              ; EH Cookie Offset
                 DD    0fffffffeH       ; previous try level for outer block
                 DD    FLAT:$LN19@main ; outer block filter
                 DD    FLAT:$LN9@main  ; outer block handler
                 DD    00H              ; previous try level for inner block
                 DD    FLAT:$LN18@main ; inner block filter
                 DD    FLAT:$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
```

```
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored pointer to scope
    table
    xor    eax, ebp                                ; ebp ^ security_cookie
    push   eax
    lea    eax, DWORD PTR __$SEHRec$[ebp+8]        ;
    pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block,
    setting previous try level=0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block,
    setting previous try level=1
    push   OFFSET $SG85497 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867 ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set
    previous try level back to 0
    jmp    SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T3[ebp], eax
    cmp    DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN5@main
```

```
    mov    DWORD PTR tv72[ebp], 1
    jmp    SHORT $LN6@main
$LN5@main:
    mov    DWORD PTR tv72[ebp], 0
$LN6@main:
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret    0

; inner block handler:
$LN13@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting
    previous try level back to 0
$LN2@main:
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
    previous try level back to -2
    jmp    SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN19@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret    0

; outer block handler:
$LN9@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting
    previous try level back to -2
$LN7@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
```

```
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_code$ = 8  ; size = 4
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push   OFFSET $SG85488 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
    jmp    SHORT $LN3@filter_use
$LN2@filter_use:
    push   OFFSET $SG85490 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP
```

Die Bedeutung des *Cookies* ist wie folgt: Der `Cookie Offset` ist die Differenz zwischen der Adresse des gespeicherten EBP-Wertes auf dem Stack und des $EBP \oplus security_cookie$-Werts auf dem Stack. Der `Cookie XOR Offset` ist eine zusätzliche Differenz zwischen dem $EBP \oplus security_cookie$-Wert und was auf dem Stack gespeichert ist.

Wenn diese Gleichung nicht richtig ist, wird der Prozess aufgrund eines korrupten Stack angehalte.

$$security_cookie \oplus (CookieXOROffset + address_of_saved_EBP) == \\ stack[address_of_saved_EBP + CookieOffset]$$

Wenn der `Cookie Offset` gleich -2 ist, impliziert dies, dass er nicht vorhanden ist.

Windows x64

Wie man sich vielleicht denken kann, ist es nicht sehr schnell bei jedem Funktionsprolog einen SEH-Frame aufzubauen. Ein weiteres Geschwindigkeitsproblem ist das

häufige Ändern des *previous try level*-Werts während der Ausführung einer Funktion.

Also haben sich die Dinge in x64 komplett geändert: alle Zeiger auf einen `try`-Block, Filter und Handler-Funktionen sind im einem PE-Segment `.pdata` gesichert. Von hier nimmt die BS-Ausnahmebehandlung alle Informationen.

Hier sind zwei Beispiele aus dem letzten Abschnitt, für x64 kompiliert:

Listing 6.33: MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, can''t
    recover'
        call    printf
        npad    1 ; align next label
$LN8@main:
```

```
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
_TEXT   ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x  ENDS
```

Listing 6.34: MSVC 2012

```
$SG86277 DB     'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB     'yes, that is our exception', 0aH, 00H
$SG86281 DB     'not our exception', 0aH, 00H
$SG86288 DB     'hello!', 0aH, 00H
$SG86290 DB     '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB     'access violation, can''t recover', 0aH, 00H
$SG86294 DB     'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD  imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
```

```
$unwind$filter_user_exceptions DD 020601H
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s
    crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main:
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t
    recover'
        call    printf
        npad    1 ; align next label
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
        call    printf
        npad    1 ; align next label
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
```

```
main    ENDP

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$:
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1 ; align next label
$LN8@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$1 ENDP
text$x  ENDS

_TEXT   SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
        call    printf
```

```
        mov     eax, 1
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use:
        lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT   ENDS
```

In [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [52]gibt es eine Reihe weiterer, detaillierte Information über dieses Thema.

Neben den Ausnahme-Informationen, beinhaltet .pdata die Adressen von fast allen Funktionsbeginn- und enden, da dies für Tools die für automatische Analysen nützlich sein kann.

Mehr über SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)][53], [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)] [54].

6.5.4 Windows NT: Kritischer Abschnitt

Kritische Abschnitte sind in jedem BS sehr wichtig bei Multithread-Umgebungen. Der Zweck besteht darin, einen exklusiven Zugriff auf eine Ressource zu garantieren, während andere Threads oder Interrupts blockiert sind.

Nachfolgend, wie eine CRITICAL_SECTION-Struktur unter Windows NT deklariert wird:

Listing 6.35: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    // The following three fields control entering and exiting the critical
    // section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
```

[52] Auch verfügbar als http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf
[53] Auch verfügbar als http://www.microsoft.com/msj/0197/Exception/Exception.aspx
[54] Auch verfügbar als http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf

```
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Nachfolgend wird gezeigt, wie die Funktion EnterCriticalSection() funktioniert:

Listing 6.36: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD:
                mov     eax, large fs:18h
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
                pop     ebp
                retn    4

... und so weiter
```

Die wichtigste Funktion in diesem Code-Fragment ist BTR (nach dem vorangehenden LOCK):

Das nullte Bit wird im CF-Flag gesichert und im Speicher zurückgesetzt. Dies ist eine Atomare Operation und blockiert alle Zugriffe der CPU auf diesen Teil des Speichert (siehe LOCK vor der BTR-Anweisung). Wenn das Bit in LockCount 1 ist, wird es zurückgesetzt und von der Funktion zurückgekehrt: die CPU befindet sich nun um Kritischen Abschnitt.

Wenn nicht, wurde der Kritische Abschnitt bereits von einem anderen Thread betreten, also muss gewartet werden. Das Warten wird durch die Funktion WaitForSingleObject() realisiert.

Hier nun, wie die Funktion LeaveCriticalSection() funktioniert:

Listing 6.37: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
                lea     edi, [esi+4]    ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... und so weiter
```

XADD bedeutet „exchange and add“.

In diesem Fall wird 1 zu LockCount addiert, während der ursprüngliche Wert von LockCount im EBX-Register gesichert wird. Der Wert in EBX wird durch aufeinander folgende INC EBX-Anweisungen inkrementiert und wird damit gleich dem aktualisierten Wert von LockCount.

Diese Operation ist atomar, da sie ebenfalls mit LOCK eingeleitet wird und so alle anderen CPUs oder CPU-Kerne des Systems für den Zugriff auf diesen Speicherbereich blockiert werden.

Das vorangehende LOCK ist sehr wichtig:

ohne diese Anweisung können zwei Threads die auf unterschiedlichen CPUs oder CPU-Kernen laufen, versuchen den Kritischen Abschnitt zu betreten und den Wert im Speicher zu verändern. Diese kann zu einem nicht-deterministischen Verhalten führen.

Kapitel 7

Tools

> Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. However, for our freedom's sake, we need free (libre) reverse engineering tools to replace the proprietary tools described in this book.
>
> Richard M. Stallman

7.1 Binäre Analyse

Tools die genutzt werden können, wenn kein Prozess gestartet wurde.

- (kostenlos, Open Source) *ent*[1]: Entropie-Analyse-Tool. Mehr über Entropie: **??** on page ??.
- *Hiew*[2]: für kleinere Modifikationen von Code in Binärdateien. Beinhaltet einen Assembler / Dissassembler.
- (kostenlos, Open Source) *GHex*[3]: Einfacher Hex-Editor für Linux.
- (kostenlos, Open Source) *xxd* und *od*: Standard UNIX-Tools für Dumping.
- (kostenlos, Open Source) *strings*: *NIX-Tool für das Suchen von ASCII-Zeichenketten in Binärdateien, inklusive ausführbaren Dateien. Sysinternals hat eine Alternative[4] die Wide-Charakter-Zeichenketten unterstützt (UTF-16, unter Windows weit verbreitet).
- (kostenlos, Open Source) *Binwalk*[5]: Analyse von Firmware-Images.

[1] http://www.fourmilab.ch/random/
[2] hiew.ru
[3] https://wiki.gnome.org/Apps/Ghex
[4] https://technet.microsoft.com/en-us/sysinternals/strings
[5] http://binwalk.org/

- (kostenlos, Open Source) *binary grep*: ein kleines Tool um jede Byte-Sequenz in einer großen Anzahl von Dateien zu suchen, inklusive nicht-ausführbaren Dateien: GitHub. Es gibt auch rafind2 in rada.re mit dem gleichen Verwendungszweck.

7.1.1 Disassembler

- *IDA*. Eine ältere Freeware-Version ist online erhältlich [6]. Wichtige Tastenkombinationen: .4.1 on page 686
- *Binary Ninja*[7]
- (kostenlos, Open Source) *zynamics BinNavi*[8]
- (kostenlos, Open Source) *objdump*: Einfaches Kommandozeilen-Tool für Dumping und zum disassemblieren.
- (kostenlos, Open Source) *readelf*[9]: Gibt Informationen über ELF-Dateien aus.

7.1.2 Decompiler

Es gibt lediglich einen bekannten, öffentlich verfügbaren Decompiler für C-Code in hoher Qualität: *Hex-Rays*:
hex-rays.com/products/decompiler/

Mehr darüber: **??** on page ??.

7.1.3 Vergleichen von Patches

Diese Tools können genutzt werden wenn die Original-Version einer ausführbaren Datei mit einer veränderten Version verglichen werden soll, oder um herauszufinden was verändert wurde und warum.

- (kostenlos) *zynamics BinDiff*[10]
- (kostenlos, Open Source) *Diaphora*[11]

7.2 Live-Analyse

Tools die im Live-System oder auf laufende Prozesse angewandt werden können.

[6] hex-rays.com/products/ida/support/download_freeware.shtml
[7] http://binary.ninja/
[8] https://www.zynamics.com/binnavi.html
[9] https://sourceware.org/binutils/docs/binutils/readelf.html
[10] https://www.zynamics.com/software.html
[11] https://github.com/joxeankoret/diaphora

7.2.1 Debugger

- (kostenlos) *OllyDbg*. Sehr populärer user-mode Debugger für die Win32-Architektur[12]. Wichtige Tastenkombinationen: .4.2 on page 687
- (kostenlos, Open Source) *GDB*. Nicht sehr populärer Debugger unter Reverse Engineers, da eher für Programmierer gemacht. Einige Kommandos: .4.5 on page 687. Es gibt eine grafische Oberfläche für GDB, "GDB dashboard"[13].
- (kostenlos, Open Source) *LLDB*[14].
- *WinDbg*[15]: Kernel-Debugger für Windows.
- *IDA* hat einen internen Debugger.
- (kostenlos, Open Source) *Radare* AKA rada.re AKA r2[16]. Es existiert auch eine GUI: *ragui*[17].
- (kostenlos, Open Source) *tracer*. Der Autor benutzt oft *tracer* [18] anstatt Debugger.

  Der Autor dieses Buchs hat irgendwann aufgehört Debugger zu nutzen, da alles was er von diesen brauchte, die Funktionsargumente während der Ausführung oder die Zustände der Register an einem bestimmten Punkt, waren. Jedes Mal den Debugger zu starten ist zu aufwändig, deswegen entstand das kleine Tool *tracer*. Es funktioniert in der Kommandozeile und erlaubt es Funktionsausführungen abzufangen, Breakpoints an beliebigen Stellen zu setzen und Register-Zustände zu lesen und ändern.

  tracer wird nicht weiterentwickelt, weil es als Demonstrationstool für dieses Buch entstand und nicht als Tool für den Alltag.

7.2.2 Tracen von Bibliotheksaufrufen

ltrace[19].

7.2.3 Tracen von Systemaufrufe

strace / dtruss

Dies zeigt welche Systemaufrufe (syscalls(6.3 on page 598)) vom aktuellen Prozess aufgerufen werden.

Zum Beispiel:

[12] ollydbg.de
[13] https://github.com/cyrus-and/gdb-dashboard
[14] http://lldb.llvm.org/
[15] https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
[16] http://rada.re/r/
[17] http://radare.org/ragui/
[18] yurichev.com
[19] http://www.ltrace.org/

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or ↙
    ↳ directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↙
    ↳ \1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"..., ↙
    ↳ 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ↙
    ↳ = 0xb75b3000
```

Mac OS X hat dtruss für den Selben Verwendungszweck.

Cygwin beinhaltet ebenso strace, funktioniert aber soweit bekannt nur mit .exe-Dateien die für die Cygwin-Umgebung kompiliert wurden.

7.2.4 Netzwerk-Analyse (Sniffing)

Sniffing ist das Abfangen einiger Informationen die interessant sein könnten.

(kostenlos, Open Source) *Wireshark*[20] für Netzwerk-Analyse. Stellt ebenfalls die Möglichkeit USB-Schnittstellen zu analysieren[21].

Wireshark hat einen jüngeren (oder älteren) Bruder *tcpdump*[22], bei dem es sich um ein simples Kommandozeilen-Tool handelt.

7.2.5 Sysinternals

(kostenlos) Sysinternals (entwickelt von Mark Russinovich) [23]. Zumindest die folgenden Tools sind wichtig und wert sich damit zu beschäftigen: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

7.2.6 Valgrind

(kostenlos, Open Source) ein mächtiges Tool um Speicherlecks zu finden: http://valgrind.org/. Wegen des ausgeklügelten JIT-Mechanismus wird Valgrind oft als Framework für andere Tools genutzt.

7.2.7 Emulatoren

- (kostenlos, Open Source) *QEMU*[24]: Emulator für verschiedene CPUs und Architekturen.

---

[20] https://www.wireshark.org/
[21] https://wiki.wireshark.org/CaptureSetup/USB
[22] http://www.tcpdump.org/
[23] https://technet.microsoft.com/en-us/sysinternals/bb842062
[24] http://qemu.org

- (kostenlos, Open Source) *DosBox*[25]: MS-DOS-Emulator, meist genutzt für Retro-Gaming.
- (kostenlos, Open Source) *SimH*[26]: Emulator für ältere Computer, Mainframes, etc.

7.3 Andere Tools

Microsoft Visual Studio Express [27]: Abgespeckte, freie Variante von Visual Studio, praktisch für einfache Experimente.

Einige nützliche Optionen: .4.3 on page 687.

Es gibt eine Website die "Compiler Explorer" heißt und es erlaubt kleine Code-Teile zu kompilieren und den Output verschiedener GCC-Versionen und Architekturen anzusehen (zumindest x86, ARM, MIPS): `http://godbolt.org/`—Ich hätte es für dieses Buch selber genutzt wenn ich davon gewusst hätte!

7.3.1 Rechner

Gute Rechner für Reverse Engineering sollten zumindest Unterstützung für Dezimal, Hexadezimal und binär-Basen, sowie wichtige Operationen wie XOR oder Schiebeoperationen haben.

- IDA hat einen eingebauten Rechner ("?").
- rada.re hat *rax2*.
- `https://yurichev.com/progcalc/`
- Als letzte Rettung hat der Standard-Rechner von Windows einen Programmierer-Modus.

7.4 Fehlt etwas?

Wenn Sie ein gutes Tool kennen, was hier nicht aufgelistet ist, schreiben Sie mir: `my emails`.

[25] `https://www.dosbox.com/`
[26] `http://simh.trailing-edge.com/`
[27] visualstudio.com/en-US/products/visual-studio-express-vs

Kapitel 8

Beispiele für das Reverse Engineering proprietärer Dateiformate

8.1 Einfache XOR Verschlüsselung

8.1.1 Einfachste XOR-Verschlüsselung überhaupt

Ich habe einmal eine Software gesehen, bei der alle Debugging-Ausgaben mit XOR mit dem Wert 3 verschlüsselt wurden. Mit anderen Worten, die beiden niedrigsten Bits aller Buchstaben wurden invertiert.

"Hello, world" wurde zu "Kfool/#tlqog":

Listing 8.1: Python

```
#!/usr/bin/python

msg="Hello, world!"

print "".join(map(lambda x: chr(ord(x)^3), msg))
```

Das ist eine ziemlich interessante Verschlüsselung (oder besser eine Verschleierung), weil sie zwei wichtige Eigenschaften hat: 1) es ist eine einzige Funktion zum Verschlüsseln und entschlüsseln, sie muss nur wiederholt angewendet werden 2) die entstehenden Buchstaben befinden sich im druckbaren Bereich, also die ganze Zeichenkette kann ohne Escape-Symbole im Code verwendet werden.

Die zweite Eigenschaft nutzt die Tatsache, dass alle druckbaren Zeichen in Reihen organisiert sind: 0x2x-0x7x, und wenn die beiden niederwertigsten Bits invertiert werden, wird der Buchstabe um eine oder drei Stellen nach links oder rechts *verschoben*, aber niemals in eine andere Reihe:

```
Characters in the coded character set ascii.
------------------------------------------------------------------------
       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
    0x C-@ C-a C-b C-c C-d C-e C-f C-g C-h TAB C-j C-k C-l RET C-n C-o
    1x C-p C-q C-r C-s C-t C-u C-v C-w C-x C-y C-z ESC C-\ C-] C-^ C-_
    2x     !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
    3x 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
    4x @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
    5x P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
    6x `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
    7x p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~   DEL
```

Abbildung 8.1: 7-Bit ASCII[1] Tabelle in Emacs

…mit dem Zeichen 0x7F als einziger Ausnahme.

Im Folgenden werden also beispielsweise die Zeichen A-Z *verschlüsselt*:

```
#!/usr/bin/python

msg="@ABCDEFGHIJKLMNO"

print "".join(map(lambda x: chr(ord(x)^3), msg))
```

Ergebnis: `CBA@GFEDKJIHONML`.

Es sieht so aus als würden die Zeichen "@" und "C" sowie "B" und "A" vertauscht werden.

Hier ist noch ein interessantes Beispiel, in dem gezeigt wird, wie die Eigenschaften von XOR ausgenutzt werden können: Exakt den gleichen Effekt, dass druckbare Zeichen auch druckbar bleiben, kann man dadurch erzielen, dass irgendeine Kombination der niedrigsten vier Bits invertiert wird.

8.2 Weiterführende Literatur

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.

How to Hack an Expensive Camera and Not Get Killed by Your Wife.

Kapitel 9

Dynamic Binary Instrumentation (DBI)

DBI-Werkzeuge können als sehr fortschrittliche und schnelle Debugger angesehen werden.

Kapitel 10

Weitere Themen

10.1 Nutzen von IMUL anstatt MUL

Beispiele wie Listing.**??** in denen zwei vorzeichenlose Werte miteinander multipliziert werden, werden zu Listing.**??** kompiliert, so dass IMUL statt MUL genutzt wird.

Dies ist eine wichtige Eigenschaft der MUL- und IMUL-Anweisung. Zunächst produzieren beide einen 64-Bit-Wert wenn zwei 32-Bit-Werte miteinander multipliziert werden, oder einen 128-Bit-Wert wenn zwei 64-Bit-Werte miteinander multipliziert werden (größtes mögliches Produkt in 32-Bit-Umgebungen ist 0xffffffff*0xffffffff=0xfffffffe00000001). Der C/C++-Standard kennt keine Möglichkeit auf die höherwertige Hälfte eines Ergebnisses zuzugreifen und ein Produkt hat immer die gleiche Größe wie die Faktoren. Beide Anweisungen MUL und IMUL arbeiten auf die gleiche Weise wenn die höherwertige Hälfte ignoriert wird. Das heißt die niederwertigere Hälfte ist die gleiche. Dies ist eine wichtige Eigenschaft der Repräsentation von vorzeichenbehafteten Zahlen im „Zweierkomplements".

Somit kann der C/C++-Compiler jede dieser Anweisungen nutzen.

Die IMUL-Anweisung ist jedoch vielseitiger als MUL weil sie jedes Register als Quelle akzeptiert, während MUL einen der Faktoren in den Registern AX, EAX oder RAX erwartet. Des weiteren sichert MUL das Ergebnis in dem EDX:EAX Paar in einer 32-Bit-Umgebung oder in RDX:RAX in einer 64-Bit-Umgebung. Die Anweisung berechnet also immer das gesamte Ergebnis. Im Gegensatz dazu ist es möglich beim Nutzen von IMUL statt eines Paares von Zielregistern ein einzelnes Register anzugeben. Die CPU wird dann lediglich den niederwertigen Teil berechnen, was zu einer höheren Geschwindigkeit führt [siehe Torborn Granlund, *Instruction latencies and throughput for AMD and Intel x86 processors*[1]).

Aus diesen Gründen ist es möglich, dass ein C/C++-Compilers öfter IMUL-Anweisungen als MUL nutzt.

Trotzdem ist es möglich mit intrinsischen Funktionen (Intrinsics) des Compilers vorzeichenlose Multiplikationen durchzuführen und das *volle* Ergebnis zu erhalten. Dies

[1] http://yurichev.com/mirrors/x86-timing.pdf]

wird manchmal *erweiterte Multiplikation* genannt. MSVC hat Intrinsics zu diesem Zweck die *__emul*[2] und *_umul128*[3] genannt werden. GCC stellt einen *__int128*-Datentyp zur Verfügung und 64-Bit-Faktoren werden zuerst auf 128-Bit erweitert, Anschließend wird das Produkt in einem anderen *__int128* gesichert. Das Ergebnis ist um 64-Bit nach rechts geshiftet um die höherwertigen Hälfte des Ergebnisses zu erhalten[4].

10.1.1 MulDiv()-Funktion in Windows

Windows hat eine MulDiv()-Funktion [5], welche die Multiplikation und Division vereint und zwei 32-Bit-Integer in einen temporären 64-Bit-Wert speichert. Anschließend findet eine Division durch eine dritte 32-Bit-Integerzahl statt. Dies ist einfacher als zwei Compiler-Intrinsics zu nutzen, weswegen die Microsoft-Entwickler diese spezielle Funktion dafür einführten. Gemessen an der Häufigkeit der Nutzung ist dies eine populäre Funktion.

10.2 Patchen von ausführbaren Dateien

10.2.1 x86-Code

Häufige Aufgaben beim Patchen sind:

- Eine der häufigsten Aufgaben ist das Deaktivieren bestimmter Anweisungen. Oft wird dies durch Austauschen des Bytes durch `0x90` (NOP).
- Bedingte Sprünge, die den Opcode wie `74 xx` (`JZ`) haben, können durch NOPs ersetzt werden.

  Es ist möglich alle bedingten Sprünge zu deaktivieren, in dem eine 0 in das zweite Byte geschrieben wird (*Sprung-Offset*).
- Eine weitere häufige Aufgabe ist es einen bedingten Sprung immer ausführen zu lassen: dies kann durch Schreiben von `0xEB`, was für `JMP` steht, anstatt des Opcodes erreicht werden.
- Die Ausführung einer Funktion kann deaktiviert werden, wenn RETN (0xC3) an den Anfang geschrieben wird. Dies gilt für alle Funktionen außer `stdcall` (6.1.2 on page 580). Um `stdcall`-Funktionen zu patchen muss die Anzahl der Argumente bekannt sein (zum Beispiel durch Finden der RETN-Anweisung in der Funktion) und die RETN-Anweisung mit einem 16-Bit-Argument (0xC2) angewendet werden.
- Manchmal muss eine deaktivierte Funktion den Wert 0 oder 1 zurückgeben. Dies kann durch `MOV EAX, 0` oder `MOV EAX, 1` erreicht werden, was aber relativ ausführlich ist. Ein besserer Weg ist `XOR EAX, EAX` (2 Byte `0x31 0xC0`) oder `XOR EAX, EAX / INC EAX` (3 Byte `0x31 0xC0 0x40`).

[2] https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx
[3] https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx
[4] Example: http://stackoverflow.com/a/13187798
[5] https://msdn.microsoft.com/en-us/library/windows/desktop/aa383718(v=vs.85).aspx

Eine Software kann gegen Manipulation geschützt sein.

Dieser Schutz ist häufig realisiert indem der ausführbare Code gelesen und ein passende Checksumme errechnet wird. Aus diesem Grund muss der Code gelesen werden bevor die Schutzfunktion aktiviert wird. Die Stelle kann durch setzen eines Breakpoints beim Lesen von Speicher herausgefunden werden.

tracer hat für diesen Zweck die BPM-Option.

Die Relocs (6.5.2 on page 615) in ausführbaren PE-Dateien sollten nicht verändert werden, da der Windows-Laser den neuen, veränderten Code möglicherweise überschreibt. (In Hiew sind die Stellen grau markiert, zum Beispiel: Abb.1.21).

Eine Möglichkeit ist es Sprünge zu schreiben, welche die Relocs umgehen oder die Reloc-Tabelle muss editiert werden.

10.3 Statistiken von Funktionsargumenten

Ich war immer sehr daran interessiert welches die durchschnittliche Anzahl von Argumenten der einzelnen Funktionen ist.

Dazu wurden viele Windows 7 32-Bit-DLLs analysiert (crypt32.dll, mfc71.dll, msvcr100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll), da diese die stdcall-Konvention nutzen, was es einfach macht das Ergebnis des Disassemblers mit grep nach `RETN X` zu durchsuchen.

- keine Argumente: $\approx 29\%$
- 1 Argument: $\approx 23\%$
- 2 Argumente: $\approx 20\%$
- 3 Argumente: $\approx 11\%$
- 4 Argumente: $\approx 7\%$
- 5 Argumente: $\approx 3\%$
- 6 Argumente: $\approx 2\%$
- 7 Argumente: $\approx 1\%$

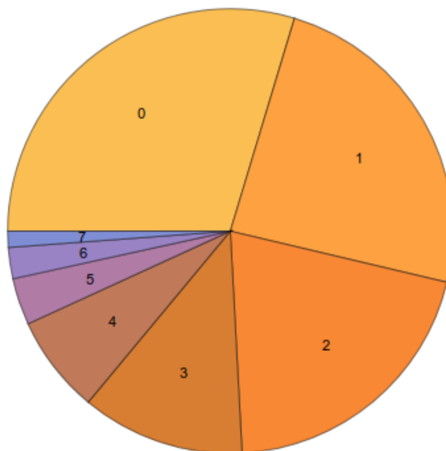


Abbildung 10.1: Statistiken von Funktionsargumenten

Das Ergebnis ist stark vom Programmierstil abhängig und kann bei anderen Programmen deutlich anders ausfallen.

10.4 Intrinsische Compiler-Funktionen

Dabei handelt es sich um spezielle Funktionen eines Compilers, die nicht in der Standard-Bibliothek enthalten sind. Der Compiler generiert einen spezifischen Maschinencode anstatt ihn aufzurufen. Dies ist häufig eine Pseudofunktion für eine spezielle CPU-Anweisung.

Beispielsweise gibt es keine zyklische Schiebe-Anweisungen in C/C++-Sprachen, in den meisten CPUs sind sie jedoch vorhanden. Um dem Programmierer das Leben einfacher zu machen hat zumindest MSVC die Pseudofunktionen *_rotl()* und *_rotr()*[6] welche vom Compiler direkt in die ROL/ROR x86-Anweisungen übersetzt werden.

Ein anderes Beispiel sind Funktionen die SSE-Anweisungen direkt im Code umwandeln.

Eine vollständige Liste von intrinsischen Funktionen in MSVC ist hier zu finden: MSDN.

[6] MSDN

10.5 Compiler Anomalien

10.5.1 Oracle RDBMS 11.2 und Intel C++ 10.1

Der Intel C++ 10.1-Compiler, der für Oracle RDBMS 11.2 für Linux 86 genutzt wurde, kann zwei JZ in einer Reihe ausgeben. Es gibt keine Referenz zum zweiten JZ. Das zweite ist also ohne Bedeutung.

Listing 10.1: kdli.o from libserver11.a

```
.text:08114CF1                   loc_8114CF1: ;
   CODE XREF: __PGOSF539_kdlimemSer+89A
.text:08114CF1                                ; __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08              mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14           movzx   edx, byte ptr [eax+14h]
.text:08114CF8 F6 C2 01              test    dl, 1
.text:08114CFB 0F 85 17 08 00 00     jnz     loc_8115518
.text:08114D01 85 C9                 test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00     jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00     jz      loc_8115518
.text:08114D0F 8B 53 08              mov     edx, [ebx+8]
.text:08114D12 89 55 FC              mov     [ebp+var_4], edx
.text:08114D15 31 C0                 xor     eax, eax
.text:08114D17 89 45 F4              mov     [ebp+var_C], eax
.text:08114D1A 50                    push    eax
.text:08114D1B 52                    push    edx
.text:08114D1C E8 03 54 00 00        call    len2nbytes
.text:08114D21 83 C4 08              add     esp, 8
```

Listing 10.2: from the same code

```
.text:0811A2A5                   loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5                                ; kdliSerLengths+1C1
.text:0811A2A5 8B 7D 08              mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10              mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14           movzx   edx, byte ptr [edi+14h]
.text:0811A2AF F6 C2 01              test    dl, 1
.text:0811A2B2 75 3E                 jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01              and     eax, 1
.text:0811A2B7 74 1F                 jz      short loc_811A2D8
.text:0811A2B9 74 37                 jz      short loc_811A2F2
.text:0811A2BB 6A 00                 push    0
.text:0811A2BD FF 71 08              push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF        call    len2nbytes
```

Dies ist vermutlich ein Fehler im Codegenerator der während der Tests nicht gefunden wurde. Der resultierende Code funktioniert trotzdem.

10.5.2 MSVC 6.0

Gerade in einem altem Code gefunden:

```
                  fabs
```

```
                 fild    [esp+50h+var_34]
                 fabs
                 fxch    st(1) ; erste Anweisung
                 fxch    st(1) ; zweite Anweisung
                 faddp   st(1), st
                 fcomp   [esp+50h+var_3C]
                 fnstsw  ax
                 test    ah, 41h
                 jz      short loc_100040B7
```

Die erste FXCH-Anweisung tauscht ST(0) und ST(1), die zweite tu das gleiche, also haben beide zusammen keine Wirkung. Das Programm nutzt MFC42.dll, also könnte es sich bei dem Compiler im MSVC 6.0, 5.0 oder eventuell MSVC 4.2 aus den 1990ern handeln.

10.5.3 Zusammenfassung

Andere Compiler-Anomalien in diesem Buch: 1.21.2 on page 368, **??** on page ??, **??** on page ??, 1.20.7 on page 352, 1.14.4 on page 169, 1.21.5 on page 390.

Diese Beispiele werden in diesem Buch gezeigt, um zu verdeutlichen, das solche Fehler in den Compilern möglich sind und es gelegentlich keinen Sinn ergibt sich den Kopf darüber zu zerbrechen warum der Compiler diesen „seltsamen“ Code erzeugte.

10.6 Itanium

Auch wenn fast gescheitert, ist der Intel Itanium (IA64) eine sehr interessante Architektur.

Während OOE[7]-CPUs entscheiden wie die Anweisungen neu organisiert werden und diese parallel ausführen, war EPIC[8] ein Versuch diese Entscheidung dem Compiler zu überlassen: das Gruppieren der Anweisungen soll während des Kompilierens erfolgen.

Dies führte zu einer berüchtigten Komplexität der Compiler.

Hier ist ein Beispiel von IA64-Code, ein einfacher kryptografischer Algorithmus aus dem Linux-Kernel:

Listing 10.3: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
```

[7]Out-of-Order Execution
[8]Explicitly Parallel Instruction Computing

```
        struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];

        n = TEA_ROUNDS;

        while (n-- > 0) {
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

Nachfolgend das Ergebnis des Compilers:

Listing 10.4: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

```
0090|                   tea_encrypt:
0090|08 80 80 41 00 21      adds r16 = 96, r32              // ptr to ctx->⤦
    ↳ KEY[2]
0096|80 C0 82 00 42 00      adds r8 = 88, r32               // ptr to ctx->⤦
    ↳ KEY[0]
009C|00 00 04 00            nop.i 0
00A0|09 18 70 41 00 21      adds r3 = 92, r32               // ptr to ctx->⤦
    ↳ KEY[1]
00A6|F0 20 88 20 28 00      ld4 r15 = [r34], 4              // load z
00AC|44 06 01 84            adds r32 = 100, r32;;           // ptr to ctx->⤦
    ↳ KEY[3]
00B0|08 98 00 20 10 10      ld4 r19 = [r16]                 // r19=k2
00B6|00 01 00 00 42 40      mov r16 = r0                    // r0 always ⤦
    ↳ contain zero
00BC|00 08 CA 00            mov.i r2 = ar.lc                // save lc ⤦
    ↳ register
00C0|05 70 00 44 10 10
     9E FF FF FF 7F 20      ld4 r14 = [r34]                 // load y
00CC|92 F3 CE 6B            movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00      nop.m 0
00D6|50 01 20 20 20 00      ld4 r21 = [r8]                  // r21=k0
00DC|F0 09 2A 00            mov.i ar.lc = 31                // TEA_ROUNDS ⤦
    ↳ is 32
00E0|0A A0 00 06 10 10      ld4 r20 = [r3];;                // r20=k1
00E6|20 01 80 20 20 00      ld4 r18 = [r32]                 // r18=k3
00EC|00 00 04 00            nop.i 0
```

```
00F0|
00F0|                               loc_F0:
00F0|09 80 40 22 00 20      add r16 = r16, r17            // r16=sum, r17⤦
    ⤷ =TEA_DELTA
00F6|D0 71 54 26 40 80      shladd r29 = r14, 4, r21      // r14=y, r21=⤦
    ⤷ k0
00FC|A3 70 68 52            extr.u r28 = r14, 5, 27;;
0100|03 F0 40 1C 00 20      add r30 = r16, r14
0106|B0 E1 50 00 40 40      add r27 = r28, r20;;          // r20=k1
010C|D3 F1 3C 80            xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20      xor r25 = r27, r26;;
0116|F0 78 64 00 40 00      add r15 = r15, r25            // r15=z
011C|00 00 04 00            nop.i 0;;
0120|00 00 00 00 01 00      nop.m 0
0126|80 51 3C 34 29 60      extr.u r24 = r15, 5, 27
012C|F1 98 4C 80            shladd r11 = r15, 4, r19      // r19=k2
0130|0B B8 3C 20 00 20      add r23 = r15, r16;;
0136|A0 C0 48 00 40 00      add r10 = r24, r18            // r18=k3
013C|00 00 04 00            nop.i 0;;
0140|0B 48 28 16 0F 20      xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00      xor r22 = r23, r9
014C|00 00 04 00            nop.i 0;;
0150|11 00 00 00 01 00      nop.m 0
0156|E0 70 58 00 40 A0      add r14 = r14, r22
015C|A0 FF FF 48            br.cloop.sptk.few loc_F0;;
0160|09 20 3C 42 90 15      st4 [r33] = r15, 4            // store z
0166|00 00 00 02 00 00      nop.m 0
016C|20 08 AA 00            mov.i ar.lc = r2;;            // restore lc ⤦
    ⤷ register
0170|11 00 38 42 90 11      st4 [r33] = r14               // store y
0176|00 00 00 02 00 80      nop.i 0
017C|08 00 84 00            br.ret.sptk.many b0;;
```

Zunächst sind alle IA64-Anweisungen in Pakete von 3 Anweisungen zusammengefasst.

Jedes Paket hat eine Größe von 16 Byte (128 Bit) und besteht aus Template-Code (5 Bit) und drei Anweisungen (je 41 Bit).

IDA zeigt die Pakete als 6+6+4 Byte, das Muster ist leicht zu erkennen.

Alle drei Anweisungen von jedem Paket wird in der Regel gleichzeitig ausgeführt, außer eine der Anweisungen enthält ein „Stop-Bit".

Vermutlich haben die Intel- und HP-Ingenieure Statistiken über die am meisten verwendeten Anweisungsmuster erhoben und entschieden die Pakettypen zu erstellen (AKA „Templates"): ein Paket-Code definiert den Anweisungstyp im Paket. Es existieren 12 von ihnen.

Beispielsweise ist der nullte Pakettyp `MII`, was impliziert, dass die erste Anweisung Speicher (Lesen oder Schreiben) ist und die zweite und dritte jeweils eine Integer-Anweisung ist.

Ein weiteres Beispiel ist das Paket vom Typ `0x1d`: `MFB`: die erste Anweisung ist be-

trifft wieder den Speicher (Lesen oder Schreiben), die zweite eine Fließkomma (FPU Anweisung) und die dritte ein Springbefehl.

Wenn der Compiler keine passende Anweisung für den entsprechenden Paketplatz finden kann, ist es möglich, dass er ein NOP einfügt: man kann hier die `nop.i`-Anweisung (NOP anstelle einer Integer-Anweisung) oder `nop.m` (anstelle einer Speicheroperation) sehen .

NOPs werden automatisch eingefügt wenn mit Assembler gearbeitet wird.

Dies ist nicht alles: Pakete können ebenfalls grupiert werden.

Jedes Paket kann ein „Stop-Bit" enthalten, so dass alle aufeinander folgenden Pakete mit einem terminierenden Paket (mit „Stop-Bit") gleichzeitig verarbeitet werden können.

In der Praxis kann Itanium 2 gleichzeitig zwei Pakete ausführen, was zu sechs Anweisungen führt.

Also kann keine der Anweisungen innerhalb einer Paket-Gruppe mit einer anderen interagieren (es kann also nicht zu Datenkonflikten kommen).

Falls sie auftreten können die Ergebnisse undefiniert sein.

Jedes Stop-Bit ist in Assembler mit zwei Semikolons (`;;`) nach de Anweisung markiert.

Die Anweisungen bei [90-ac] können also simultan ausgeführt werden: sie beeinflussen sich gegenseitig nicht. Die nächste Gruppe ist [b0-cc].

Hier ist auch das Stop-Bit bei 10c zu sehen Die nächste Anweisung bei 110 hat ebenfalls ein Stop-Bit.

Dies impliziert dass diese Anweisungen von allen anderen getrennt ausgeführt werden müssen (wie in CISC).

Außerdem ist zu sehen, dass die Anweisung nach 110 das Ergebnis der vorangehenden benutzt (Den Wert im Register r26), dementsprechend können sie nicht gleichzeitig ausgeführt werden.

Anscheinend war der Compiler nicht in der Lage einen besseren Weg zum Parallelisieren der Anweisungen zu finden, also die CPU so weit wie möglich auszulasten. Daher die vielen Stop-Bits und NOP-Anweisungen.

Manuelle Assembler-Programmierung ist ein mühsamer Job: der Programmierer muss die Anweisungen selber in Gruppen einteilen.

Der Programmierer ist immer noch in der Lage Stop-Bits zu jeder Anweisung hinzuzufügen, doch dies wird die Geschwindigkeit heruntersetzen für die Itanium gemacht wurde.

Ein interessantes Beispiel von manuellem Assembler-Code in IA64 kann im Code des Linux-Kernels gefunden werden:

`http://lxr.free-electrons.com/source/arch/ia64/lib/`.

Eine weitere Einführung für den Itanium-Assembler: [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)][9], [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)][10].

Weitere sehr interessante Itanium-Features sind *speculative execution* und das NaT („not a thing")-Bit, was in gewisser Weise NaN-Zahlen ähnelt:

MSDN

10.7 8086-Speichermodell

Wenn es um 16-Bit-Programme für MS-DOS oder Win16 geht (**??** on page ?? oder **??** on page ??), kann man sehen, dass die Zeiger aus zwei 16-Bit-Werten bestehen. Was bedeutet das? Ja, das ist wieder ein weiteres sonderbares Artefakt von MS-DOS und 8086.

8086/8088 war eine 16-Bit-CPU, war aber in der Lage 20-Bit-Adressen im RAM anzusprechen (und somit externen Speicher bis 1MB zu adressieren).

Der Adressbereich für externen Speicher ist aufgeteilt zwischen RAM (maximal 640KB), **ROM!**, Fenster für Videospeicher, EMS-Karten, etc.

Erinnern wir uns auch nochmal daran, dass der 8086/8088 der Nachfolger der 8-Bit-CPU 8080 war.

Der 8080 hat einen 16-Bit-Adressspeicher, kann also lediglich 64KB Speicher adressieren.

Möglicherweise aus Gründen der Portierung alter Software[11], kann der 8086 viele 64KB-Fenster gleichzeitig unterstützen, die sich im 1MB-Adressbereich befinden.

Dies ist eine Art Top-Level-Virtualisierung.

Alle 8086-Register sind 16-Bit breit. Um einen größeren Bereich adressieren zu können, wurden spezielle Segment-Register (CS, DS, ES, SS) eingeführt.

Jeder 20-Bit-Zeiger wird aus den Werten eines Segment-Registers und einem Adressregister-Paar (z.B. DS:BX) berechnet.

$$reale_adresse = (segment_register \ll 4) + adress_register$$

Zum Beispiel: das Grafik-Video-Speicher-Fenster (EGA[12], VGA[13]) auf alten zu IBM PC kompatiblen Rechnern hat eine Größe von 64KB.

Um darauf zuzugreifen muss der Wert 0xA000 in eines der Segment-Register geschrieben werden, zum Beispiel in DS.

Anschließend wird DS:0 das erste Byte des Video-RAM und DS:0xFFFF das letzte Byte adressieren.

[9] Auch verfügbar als http://yurichev.com/mirrors/RE/itanium.pdf
[10] Auch verfügbar als http://phrack.org/issues/57/5.html
[11] Der Autor ist sich hier jedoch nicht 100% sicher.
[12] Enhanced Graphics Adapter
[13] Video Graphics Array

Die echte Adresse auf dem 20-Bit-Adressbus ist in dem Bereich zwischen 0xA0000 und 0xAFFFF.

Das Programm kann hart-kodierte Adressen wie 0x1234 beinhalten, das BS lädt das Programm aber bei Bedarf an eine beliebige Adresse. Dazu werden die Segment-Registerwerte derart neu berechnet, dass das Programm sich nicht darum kümmern muss an welcher Stelle im RAM es sich befindet.

Jeder Zeiger in der alten MS-DOS-Umgebung besteht aus der Segmentadresse und der Adresse innerhalb des Segment, also zwei 16-Bit-Werten. 20 Bit sind hierfür genug, allerdings muss die Adresse recht oft neu berechnet werden. Mehr Informationen auf dem Stack zu übergeben schien eine bessere Speicher- / Komfort-Balance zu haben.

Übrigens: aufgrund all der vorherigen Überlegungen war es nicht mögliche Speicherblöcke zu allozieren die größer 64KB waren.

Die Segmentregister wurde beim 80286 als „Selektoren“ wieder genutzt, jedoch mit einer anderen Funktion.

Als die 80386-CPU mit größerem RAM eingeführt wurde, war MS-DOS immer noch weit verbreite, so das die DOS-Extender auftraten. Diese waren eigentlich ein Schritt zu einem „seriösen“ BS indem die CPU in den Protected Mode geschaltet wurde und sehr viel bessere Speicher-APIs für die Programme angeboten wurden, die noch unter MS-DOS liefen.

Sehr populäre Beispiele waren DOS/4GW (das Spiel DOOM wurde hierfür kompiliert), Phar Lap und PMODE.

Übrigens wurde das gleiche Adressierungsmodel für Speicher in der 16-Bit-Reihe von Windows 3.x genutzt, bevor Win32 aufkam.

10.8 Basic Block Reordering

10.8.1 Profile-guided Optimization

Diese Optimierungsmethode kann einige Einfacher Blocks zu anderen Sektionen der ausführbaren Datei verschieben.

Offensichtlich gibt es Teile einer Funktion die öfter ausgeführt werden als andere (zum Beispiel Schleifen-Rümpfe) und welche, die weniger oft ausgeführt werden (beispielsweise Fehlerberichte oder Ausnahmebehandlungen).

Der Compiler fügt Messcode in die ausführbare Datei ein. Anschließend führt der Programmierer diesen mit vielen Tests aus um Statistiken zu erstellen.

Der Compiler präpariert die ausführbare Datei mithilfe der erstellten Statistiken insofern, dass alle weniger häufige Codeteile in eine andere Sektion der Datei verschoben werden.

Als Ergebnis ist der häufig ausgeführte Funktionscode zusammengefasst, was sehr wichtig für die Ausführgeschwindigkeit und die Cachebenutzung ist.

Ein Beispiel vom Oracle RDBMS-Code, der mit dem Intel C++-Compiler übersetzt wurde:

Listing 10.5: orageneric11.dll (win32)

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue:
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error                ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...

; address 0x60B953F0

__VInfreq__skgfsync:
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ;
    "skgfsync(se=0x%x, ctx=0x%x, iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue

error:
```

```
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function called with
    invalid FIB/IOV structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

Der Abstand der Adressen zwischen diesen beiden Code-Fragmenten beträgt fast 9 MB.

Alle weniger oft ausgeführten Codeteile wurden an das Ende der Code-Sektion der DLL-Datei verschoben.

Dieser Teil der Funktion wurde vom Intel C++-Compiler mit dem `VInfreq`-Präfix markiert.

Man kann hier sehen, dass der Teil der Funktion der in die Logdatei schreibt (zum Beispiel im Falle eines Fehlers oder einer Warnung) vermutlich selten oder vielleicht gar nicht ausgeführt wurde als der Entwickler von Oracle die Statistiken erstellt hat.

Das Schreiben in log basic block gibt die Ausführkontrolle letztendlich wieder zurück an den „heißen" Teil der Funktion.

Ein weiterer „seltener" Teil ist der Einfacher Block, welcher den Fehlercode 27050 zurück gibt.

In Linux ELF-Dateien wird der selten ausgeführte Code vom Intel C++-Compiler in die separate `text.unlikely`-Sektion verschoben und der „heiße" Code in die Sektion `text.hot`.

Aus Sicht eines Reverse-Engineers kann diese Information helfen um die Funktion in den Hauptteil und den Fehlerbehandlungsteil zu unterteilen.

Kapitel 11

Bücher / Lesenswerte Blogs

11.1 Bücher und andere Materialien

11.1.1 Reverse Engineering

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

Ebenfalls das Buch von Kris Kaspersky.

11.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The “Ultimate” Anti-Debugging Reference[1]

Blogs:

- Microsoft: Raymond Chen
- nynaeve.net

[1] http://pferrie.host22.com/papers/antidebug.pdf

11.1.3 C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)[2]
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard[3]
- Agner Fog, *Optimizing software in C++* (2015)[4]
- Marshall Cline, *C++ FAQ*[5]
- Dennis Yurichev, *C/C++ programming language notes*[6]
- JPL Institutional Coding Standard for the C Programming Language[7]

11.1.4 x86 / x86-64

- Intel Handbücher[8]
- AMD Handbücher[9]
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)[10]
- Agner Fog, *Calling conventions* (2015)[11]
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

Etwas veraltet aber immer noch interessant zu lesen:

Michael Abrash, *Graphics Programming Black Book*, 1997[12] (Er ist bekannt für seine Arbeiten auf dem Gebiet der Low-Level Optimierung in Projekten wie Windows NT 3.1 und id Quake).

11.1.5 ARM

- ARM Handbücher[13]
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)

[2]Auch verfügbar als http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
[3]Auch verfügbar als http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.
[4]Auch verfügbar als http://agner.org/optimize/optimizing_cpp.pdf.
[5]Auch verfügbar als http://www.parashift.com/c++-faq-lite/index.html
[6]Auch verfügbar als http://yurichev.com/C-book.html
[7]Auch verfügbar als https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
[8]Auch verfügbar als http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
[9]Auch verfügbar als http://developer.amd.com/resources/developer-guides-manuals/
[10]Auch verfügbar als http://agner.org/optimize/microarchitecture.pdf
[11]Auch verfügbar als http://www.agner.org/optimize/calling_conventions.pdf
[12]Auch verfügbar als https://github.com/jagregory/abrash-black-book
[13]Auch verfügbar als http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)][14]
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)[15]

11.1.6 Assembler

Richard Blum — Professional Assembly Language.

11.1.7 Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] [16].

11.1.8 UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

11.1.9 Programmierung Allgemein

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002). Einige Leute sagen, die Tricks und Hacks aus diesem Buch sind heute nicht mehr relevant und haben die eigentliche Bedeutung für RISC CPUs, bei denen Verzweigungsbefehle teuer sind. Nichtsdestotrotz können diese immens hilfreich sein um Bool'sche Algebra und die damit zusammenhängende Mathematik zu verstehen.

11.1.10 Kryptografie

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) lvh, *Crypto 101*[17]
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*[18].

[14] Auch verfügbar als http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf
[15] Auch verfügbar als https://yurichev.com/ref/ARM%20Cookbook%20(1994)/
[16] Auch verfügbar als https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/
[17] Auch verfügbar als https://www.crypto101.io/
[18] Auch verfügbar als https://crypto.stanford.edu/~dabo/cryptobook/

Kapitel 12

Communities

Es gibt zwei exzellente Subreddits auf reddit.com mit RE[1]-relevanten Themen: reddit.com/r/ReverseEngineering/ und reddit.com/r/remath (für Themen mit der Schnittmenge RE und Mathematik).

Es gibt außerdem einen RE-relevanten Teil auf der Stack Exchange-Seite: reverseengineering.stackexchange.com.

Im IRC gibt es einen ##re-Kanal auf Libera.

[1] Reverse Engineering

Nachwort

12.1 Fragen?

Zögern Sie nicht dem Autor Ihre Fragen per my emails zu schicken. Haben Sie irgendwelche Vorschläge für neue Inhalte in diesem Buch? Gerne können Sie Korrekturen (auch grammatischer Art, vor allem im Englischen) zuschicken.

Der Autor arbeitet sehr viel an diesem Buch, so dass sich Seitenzahlen, Nummerierungen und so weiter schnell ändern können. Bitte beziehen Sie sich also nicht auf diese Angaben. Einfacher ist es einen Screenshot von der entsprechenden Seite zu machen und den Fehler in einer Bildbearbeitung zu markieren. Der Autor kann so den Fehler sehr viel schneller korrigieren. Wenn Sie sich mit git und LaTeX auskennen, können Sie den Fehler direkt im Quellcode ändern:

https://beginners.re/src/.

Auch wenn Sie sich nicht ganz sicher sind, teilen Sie bitte die kleinen oder großen Fehler mit, die Sie finden. Dieses Buch richtet sich speziell an Anfänger, so dass deren Meinungen und Kommentare einen entscheidenden Einfluss haben.

Anhang

.1 x86

.1.1 Terminologie

Geläufig für 16-Bit (8086/80286), 32-Bit (80386, etc.), 64-Bit.

Byte 8-Bit. Die DB Assembler-Direktive wird zum Definieren von Variablen und Arrays genutzt. Bytes werden in dem 8-Bit-Teil der folgenden Register übergeben: AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L.

Wort 16-Bit. DW Assembler-Direktive —”—. Bytes werden in dem 16-Bit-Teil der folgenden Register übergeben: AX/BX/CX/DX/SI/DI/R*W.

Doppelwort („dword“) 32-Bit. DD Assembler-Direktive —”—. Doppelwörter werden in Registern (x86) oder dem 32-Bit-Teil der Register (x64) übergeben. In 16-Bit-Code werden Doppelwörter in 16-Bit-Registerpaaren übergeben.

zwei Doppelwörter („qword“) 64-Bit. DQ Assembler-Direktive —”—. In 32-Bit-Umgebungen werden diese in 32-Bit-Registerpaaren übergeben.

tbyte (10 Byte) 80-Bit oder 10 Bytes (für IEEE 754 FPU Register).

paragraph (16 Byte) — Bezeichnung war in MS-DOS Umgebungen gebräuchlich.

Datentypen der selben Breite (BYTE, WORD, DWORD) entsprechen auch denen in der Windows API.

.1.2 npad

Dies ist ein Assembler-Makro um Labels an bestimmten Grenzen auszurichten.

Dies ist oft nützlich Labels, die oft Ziel einer Kotrollstruktur sind, wie Schleifenköpfe. Somit kann die CPU Daten oder Code sehr effizient vom Speicher durch den Bus, den Cache, usw. laden.

Entnommen von `listing.inc` (MSVC):

Dies ist übrigens ein Beispiel für die unterschiedlichen NOP-Variationen. Keine dieser Anweisungen hat eine Auswirkung, aber alle haben eine unterschiedliche Größe.

Eine einzelne Idle-Anweisung anstatt mehrerer NOPs hat positive Auswirkungen auf die CPU-Performance.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the files created
;; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
```

```
if size eq 2
  mov edi, edi
else
 if size eq 3
   ; lea ecx, [ecx+00]
   DB 8DH, 49H, 00H
 else
  if size eq 4
    ; lea esp, [esp+00]
    DB 8DH, 64H, 24H, 00H
  else
   if size eq 5
     add eax, DWORD PTR 0
   else
    if size eq 6
      ; lea ebx, [ebx+00000000]
      DB 8DH, 9BH, 00H, 00H, 00H, 00H
    else
     if size eq 7
       ; lea esp, [esp+00000000]
       DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
     else
      if size eq 8
       ; jmp .+8; .npad 6
       DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
      else
       if size eq 9
        ; jmp .+9; .npad 7
        DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
       else
        if size eq 10
         ; jmp .+A; .npad 7; .npad 1
         DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
        else
         if size eq 11
          ; jmp .+B; .npad 7; .npad 2
          DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
         else
          if size eq 12
           ; jmp .+C; .npad 7; .npad 3
           DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
          else
           if size eq 13
            ; jmp .+D; .npad 7; .npad 4
            DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24↙
↪ H, 00H
           else
            if size eq 14
             ; jmp .+E; .npad 7; .npad 5
             DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, ↙
↪ 00H, 00H, 00H
            else
             if size eq 15
```

```
                ; jmp .+F; .npad 7; .npad 6
                DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, ↙
    ↘ 00H, 00H, 00H, 00H
               else
                %out error: unsupported npad size
                .err
               endif
              endif
             endif
            endif
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

.2 Einige GCC-Bibliotheks-Funktionen

Name	Bedeutung
__divdi3	vorzeichenbehaftete Division
__moddi3	Rest (Modulo) einer vorzeichenbehafteten Division
__udivdi3	vorzeichenlose Division
__umoddi3	Rest (Modulo) einer vorzeichenlosen Division

.3 Einige MSVC-Bibliotheks-Funktionen

`ll` in Funktionsnamen steht für „long long“, z.B. einen 64-Bit-Datentyp.

Name	Bedeutung
__alldiv	vorzeichenbehaftete Division
__allmul	Multiplikation
__allrem	Rest einer vorzeichenbehafteten Division
__allshl	Schiebe links
__allshr	Schiebe links, vorzeichenbehaftet
__aulldiv	vorzeichenlose Division
__aullrem	Rest (Modulo) einer vorzeichenlosen Division
__aullshr	Schiebe rechts, vorzeichenlos

Multiplikation und Links-Schiebebefehle sind sowohl für vorzeichenbehaftete als auch vorzeichenlose Zahlen, da hier für jede Operation nur ein Befehl existiert .

Der Quellcode dieser Funktionen kann im Pfad des installierten MSVS[2], gefunden werden: `VC/crt/src/intel/*.asm`.

.4 Cheatsheets

.4.1 IDA

Wichtige Tastenkombinationen:

Taste	Bedeutung
Space	Zwischen Quellcode und grafischer Ansicht wechseln
C	zu Code konvertieren
D	zu Daten konvertieren
A	zu Zeichenkette konvertieren
*	zu Array konvertieren
U	undefinieren
O	Offset von Operanden
H	Dezimalzahl erstellen
R	Zeichen erstellen
B	Binärzahl erstellen
Q	Hexadezimalzahl erstellen
N	Identifikator umbenennen
?	Rechner
G	zu Adresse springen
:	Kommentar einfügen
Ctrl-X	Referenz zu aktueller Funktion, Variable, ... zeigen (inkl. lokalem Stack)
X	Referenz zu Funktion, Variable, ... zeigen
Alt-I	Konstante suchen
Ctrl-I	Nächstes Auftreten der Konstante suchen
Alt-B	Byte-Sequenz suchen
Ctrl-B	Nächstes Auftreten der Byte-Sequenz suchen
Alt-T	Text suchen (inkl. Anweisungen, usw.)
Ctrl-T	nächstes Aufreten des Textes suchen
Alt-P	akutelle Funktion editieren
Enter	zu Funktion, Variable, ... springen
Esc	zurückgehen
Num -	Funktion oder markierten Bereich einklappen
Num +	Funktion oder Bereich anzeigen

Das Einklappen ist nützlich um Teile von Funktionen zu verstecken, wenn bekannt ist was sie tun. dies wird genutzt imScript[3] um häufig genutzte Inline-Code-Stellen zu verstecken.

[2] Microsoft Visual Studio
[3] GitHub

.4.2 OllyDbg

Wichtige Tastenkombinationen:

Tastenkürzel	Bedeutung
F7	Schritt
F8	step over
F9	starten
Ctrl-F2	Neustart

.4.3 MSVC

. Einige nützliche Optionen die in diesem Buch genutzt werden.

Option	Bedeutung
/O1	Speicherplatz minimieren
/Ob0	Keine Inline-Erweiterung
/Ox	maximale Optimierung
/GS-	Sicherheitsüberprüfungen deaktivieren (Buffer Overflows)
/Fa(file)	Assembler-Quelltext erstellen
/Zi	Debugging-Informationen erstellen
/Zp(n)	Strukturen an n-Byte-Grenze ausrichten
/MD	ausführbare Daten nutzt `MSVCR*.DLL`

Informationen zu MSVC-Versionen: 5.1.1 on page 543.

.4.4 GCC

Einige nützliche Optionen die in diesem Buch genutzt werden.

Option	Bedeutung
-Os	Optimierung der Code-Größe
-O3	maximale Optimierung
-regparm=	Anzahl der in Registern übergebenen Argumente
-o file	Name der Ausgabedatei
-g	Debug-Informationen in der ausführbaren Datei erzeugen
-S	Assembler-Quellcode erstellen
-masm=intel	Quellcode im Intel-Syntax erstellen
-fno-inline	keine Inline-Funktionen verwenden

.4.5 GDB

Einige nützliche Optionen die in diesem Buch genutzt werden:

Option	Bedeutung
break filename.c:number	Setzen eines Breakpoints in der angegebenen Zeile
break function	Setzen eines Breakpoints in der Funktion
break *address	Setzen eines Breakpoints auf Adresse
b	—”—
p variable	Ausgabe eines Variablenwerts
run	Starten
r	—”—
cont	Ausführung fortfahren
c	—”—
bt	Stack ausgeben
set disassembly-flavor intel	Intel-Syntax nutzen
disas	disassemble current function
disas function	Funktion disassemblieren
disas function,+50	disassemble portion
disas $eip,+0x10	—”—
disas/r	mit OpCodes disassemblieren
info registers	Ausgabe aller Register
info float	Ausgabe der FPU-Register
info locals	(bekannte) lokale Variablen ausgeben
x/w ...	Speicher als 32-Bit-Wort ausgeben
x/w $rdi	Speicher als 32-Bit-Wort ausgeben an Adresse in RDI
x/10w ...	10 Speicherworte ausgeben
x/s ...	Speicher als Zeichenkette ausgeben
x/i ...	Speicher als Code ausgeben
x/10c ...	10 Zeichen ausgeben
x/b ...	Bytes ausgeben
x/h ...	16-Bit-Halbworte ausgeben
x/g ...	große (64-Bit-) Worte ausgeben
finish	bis Funktionsende fortfahren
next	Nächste Anweisung (nicht in Funktion springen)
step	Nächste Anweisung (in Funktion springen)
set step-mode on	Beim schrittweisen Ausführen keine Zeilennummerninfos nutzen
frame n	Stack-Frame tauschen
info break	Breakpoints schauen
del n	Breakpoints löschen
set args ...	Aufrufparameter setzen

Verwendete Abkürzungen

TLS Thread Local Storage 329

AKA Also Known As — auch bekannt als 40

CRT C Runtime library 14

CPU Central Processing Unit xiv

FPU Floating-Point Unit v

CISC Complex Instruction Set Computing 26

RISC Reduced Instruction Set Computing 3

GUI Graphical User Interface 607

RTTI Run-Time Type Information 579

BSS Block Started by Symbol 33

SIMD Single Instruction, Multiple Data 226

BSOD Blue Screen of Death 598

DBMS Database Management Systems xi

ISA Instruction Set Architecture 2

SEH Structured Exception Handling 622

ELF Executable and Linkable Format: In *NIX Systemen einschließlich Linux weit verbreitetes Format für ausführbare Dateien 86

TIB Thread Information Block 329

PIC Position Independent Code 600

NOP No Operation 8

BEQ (PowerPC, ARM) Branch if Equal 105

BNE (PowerPC, ARM) Branch if Not Equal 242

XOR eXclusive OR 697

RAM Random-Access Memory 88

GCC GNU Compiler Collection 5

EGA Enhanced Graphics Adapter 672

VGA Video Graphics Array 672

API Application Programming Interface 547

ASCII American Standard Code for Information Interchange

ASCIIZ ASCII Zero ( ) 103

IA64 Intel Architecture 64 (Itanium) 614

EPIC Explicitly Parallel Instruction Computing 668

OOE Out-of-Order Execution 668

STL (C++) Standard Template Library 542

VM Virtual Memory

WRK Windows Research Kernel 566

GPR General Purpose Registers 2

SSDT System Service Dispatch Table 598

RE Reverse Engineering 679

CDFS Compact Disc File System 574

CD Compact Disc

ADC Analog-to-Digital Converter 570

EOF End of File (Dateiende) 94

TBT To be Translated. The presence of this acronym in this place means that the English version has some new/modified content which is to be translated and placed right here. 122

DBI Dynamic Binary Instrumentation 546

URL Uniform Resource Locator 556

Glossar

Anti-Pattern Allgemein als schlecht erachtetes Vorgehen. 44, 82

Atomare Operation „$\alpha\tau o\mu o\varsigma$“ steht im Griechischen für „unteilbar“. Eine solche Operation wird garantiert nicht von anderen Threads unterbrochen. 653

Blatt-Funktion Eine Funktion, die keine weitere Funktion aufruft. 38, 43

callee aufgerufene Funktion. 44, 62, 108, 111, 114, 580, 582–584, 587–589

caller aufrufende Funktion. 8, 14, 40, 62, 108–110, 113, 178, 580, 581, 583, 584, 589

Dekrement Verminderung um 1. 213, 235, 525, 577

Einfacher Block Eine Instruktionsgruppe die keine Sprung- ode Verzweigungsinstruktionen enthält, und in die es auch keine Sprünge von außerhalb gibt. In IDA sieht das einfach wie eine Liste von Instruktionen ohne Leerzeilen aus. 673, 675

Endianness Byte-Reihenfolge. 29, 85, 407

GiB Gibibyte: 2^{30} oder 1024 Mebibytes oder 1073741824 Bytes. 21

Heap Üblicherweise ein großes vom Betriebssystem bereitgestelltes Stück Speicher, welches Anwendungen nach Wunsch selber unterteilen können. malloc()/free() arbeiten auf dem Heap. 41, 610, 611

Inkrement Erhöhung um 1. 22, 213, 218, 235, 525

Jump Offset Teil des Opcodes einer JMP oder Jcc Instruktion, der zur Adresse der nächsten Instruktion addiert wird um den neuen **PC!** zu berechnen. Kann auch negativ sein. 104, 150

Link Register (RISC) Ein Register, in dem üblicherweise die Rücksprungadresse gespeichert wird. So können Blatt-Funktionen aufgerufen werden ohne den Stack zu benutzen, d.h., schneller. 43

Loop unwinding Wenn ein Compiler anstatt Schleifencode für n Wiederholungen n Kopien des Schleifenkörpers erzeugt, um Instruktionen für die Schleifenverwaltung einzusparen. 216

Name Mangling Wird u.a. in C++ genutzt, wobei der Compiler den Namen einer Klasse, Methode und deren Argumenttypen in eine Zeichenkette kodiert, die als interner Name für die Funktion verwendet wird. Mehr dazu steht in **??** on page ??. 544

NaN Not a Number: Ein Spezialfall bei Gleitkommazahlen, signalisiert normalerweise einen Fehler. 272, 294, 295, 672

NOP „No operation“, Leerlaufinstruktion. 577

Padding *Padding* bedeutet wörtlich übersetzt ein Kissen auszustopfen um es in eine gewünschte Größe zu bringen. In der Informatik bedeutet es, einem Block ein paar Bytes hinzuzufügen bis dieser eine gewünschte Größe erreicht, wie z.B. 2^n Bytes.. 555

PDB (Win32) Datei mit Debuginformationen, meistens Funktionsnamen, manchmal auch Funktionsargumente und lokale Variablen. 543, 614

POKE Befehl aus der Sprache BASIC, um ein Byte an eine bestimmte Adresse zu schreiben. 578

Produkt Ergebnis einer Multiplikation. 260, 264, 663, 664

Quotient Ergebnis einer Division. 253, 256, 258, 259, 264

Reelle Zahl Zahlen die ein Gleitkomma enthalten können. In C/C++ sind das sind *float* und *double*. 253

Register Allokator Der Teil des Compilers, der lokalen Variablen CPU Register zuweist. 233, 358, 501

Stack Frame Der Teil des Stacks, der Informationen speziell zur aktuellen Funktion enthält: Lokale Variablen, Funktionsargumente, RA, etc.. 71, 109, 641

Stapel-Zeiger Ein Register das auf eine Stelle im Stack zeigt. 13, 15, 41, 47, 57, 78, 111, 580, 582–584

stdout Standardausgabe. 28, 48, 179

Thunk-Funktion Kleine Funktion mit dem einzigen Zweck, eine andere Funktion aufzurufen. 30

tracer Mein eigenes einfaches Debug-Werkzeug. Mehr dazu hier: 7.2.1 on page 657. 219–221, 548, 563, 567, 635, 665

Windows NT Windows NT, 2000, XP, Vista, 7, 8, 10. 340, 498, 598, 612, 652

xoring Im Englischen oft genutzt um die Anwendung einer XOR[4] Verknüpfungd auszudrücken. 641

[4] eXclusive OR

Index